1989

Description of SEMILOG

Ryan Stanisfer

Chul Hung

Andrew B. Whinston

Parthasarathy Bhasker

Report Number:
89-868

DESCRIPTION OF SEMILOG

Ryan Stansifer
Chul Jung
Andrew B. Whinston
Parthasarathy Bhasker

CSD-TR-868
February 1989

Description of SEMLOG

Ryan Stansifer
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907-7821

Chul Jung
Department of Management Science and Information Systems
University of Texas

Andrew B. Whinston
Department of Management Science and Information Systems
University of Texas

Parthasarathy Bhasker
Battelle Memorial Institute
Columbus, Ohio

February 27, 1989

Contents

1 Introduction

In this paper we describe a new language, SEMLOG, which integrates features from several programming paradigms. We believe this language and its extensions will be a good basis for developing management oriented information systems. SEMLOG incorporates features from object-oriented, functional and logic languages. These features are integrated so that SEMLOG is not just an amalgam of three language families. SEMLOG is also strongly typed so that through type checking unintentional deviations from the type specification are detected before the program is run.

2 AI Programming Paradigms and Languages

In this section we present an overview of the different programming paradigms that have been used for developing knowledge based applications. For each paradigm we will examine what makes it appealing for developing knowledge-based systems. We will argue the need for combining the paradigms within a single framework, and follow this with a discussion on the currently available LISP-based multiparadigm environments for building knowledge-based systems. A discussion of the limitations of these multiparadigm environments ensues.

One limitation of these multiparadigm environments — the inability to represent arbitrary relations among domain objects and reasoning based on these relations — is the forte of logic languages such as PROLOG [Clocksin 1984] and LOGIN [Aït-Kaci 1986b]. However, these languages have other inadequacies and these are discussed in an informal manner in this section. Finally, we present an overview of the attempts to provide object-oriented programming in PROLOG.

2.1 Programming Paradigms

A programming paradigm or style of programming supports the expression of a programmer's intent. A programming language is a medium in which one or more of the paradigms can be expressed. For example, logic programming is a paradigm, whereas PROLOG is a programming language that is based on that paradigm.

In this section we present an overview of different AI programming paradigms: functional, logic, and object-oriented.

Functional Paradigm

The functional style of programming uses function application and recursive function definitions as the principal means of computing. A program in a functional language consists of independent functions, each a mapping from its arguments to a result. In order to program with functions the programmer needs to start with a rich set of basic functions and then use various combining forms to define new functions in terms of the basic functions and other user defined functions. Recursion and higher-order functions are two techniques that are used in functional languages to build powerful functions.

An important notion that is associated with functional programming is that the value of the expression is determined solely by the values its constituent parts. Thus, should the same expression occur twice in the same context, it denotes the same value in both occurrences. A language that supports this property for all of its expressions is referred to as a *purely functional language*. In other words a purely functional language computes values, it does not compute for effect.

The programming language LISP has been the most popular and widely used programming language in this class of languages. Among the languages used for AI programming, LISP is by far the most popular, although PROLOG seems to be steadily gaining popularity. While the LISP language is more functional in style than many other programming languages it nevertheless provides many features that perform side effects. Thus it is no more the pure functional programming language that it was originally conceived to be.

LISP's AI popularity stems from several features: easy and flexible symbol manipulation, automatic memory management, sophisticated environments and debugging aids, and the uniform treatment of pro-

grams and data. In addition, LISP is available on a variety of computers including special-purpose LISP machines, which have been designed to execute LISP code with speed and efficiency.

Lists were the principal means of building data structures in LISP for a long time. Now many dialects of LISP, including COMMON LISP, provide facilities for creating record structures with named components. In effect, with this new facility, the user can define new record data types. When such a data type is created in LISP, constructor, access, and assignment constructs are automatically defined.

It is important to note however, that even with these new facilities, type checking in LISP continues to be done when target programs run, not when they are compiled. In other words type checking is *dynamic*, not *static*. Since there is no means for the compiler to guarantee that the program it accepts will execute with no type errors, LISP is not a *strongly typed* language.

Another functional language ML which originated as a metalanguage of Edinburgh LCF, has evolved into a programming language in its own right [Milner 1984]. In ML, functions are first-class objects, and thus they can be passed as arguments to other functions and can be the result of function evaluations. Unlike LISP, ML is a strongly typed language.

Logic Paradigm

Around 1970, Kowalski and Colmerauer were led to the fundamental idea of *programming in logic*. The idea of using a subset of first order predicate calculus as a programming language was a significant contribution, because, until about 1970, computer scientists used logic only as a specification language. However, Kowalski [Kowalski 1974] and others showed that logic has a *procedural interpretation* as well, making it possible in principle to use logic as a programming language.

The main thesis of logic programming, as expressed by [Kowalski 1979] is that an algorithm can be usefully expressed in two components: the *logic* and the *control* component. The logic component is the statement of *what* the problem is. The control part is a statement of *how* it is to be solved. The ideal goal of logic programming is that the programmer need specify only the logic component of a problem. The control should be exercised by the computer. In other words, the logic programmer should only concern himself with ensuring that the logic formulas in his program are a true reflection of the problem. This ideal has not yet been achieved. Currently programmers need to provide small but undue amounts of control information, in the form of extra-logical "features" in the language.

The programming language PROLOG [Clocksin 1984] has been the flagship of the logic paradigm. Programs in PROLOG have a declarative syntax. The Horn clauses in PROLOG programs can be read as implications, universally quantified by the variables occurring in the clause.

From a knowledge engineering and conceptual modeling perspective PROLOG's declarative representation of knowledge, the ability to represent arbitrary relations among objects, and the deductive capability are its most important assets.

The declarative syntax of PROLOG is also an important asset from a software engineering perspective. A PROLOG program is viewed as a specification of the task that is to be accomplished; there is no need to lay out the steps needed to attain the solution.

Object-Oriented Programming

The object-oriented paradigm has proven to be a valuable way of organizing programs in some task domains. In this paradigm a program consists of objects and messages. Objects represent state, and messages represent the behavior of objects.

The objects are grouped together into classes, all of which have the same structure and behavior. Behavior is invoked by sending a message. A message consists of an object to which the message is directed, the selector (the name of the behavior), and other parameters. The response to a message is determined by the class of the object to which the message is directed.

Message sending supports an important principle of programming: *data abstraction*. The key notion behind data abstraction is information hiding. In the object-oriented programming methodology modules share information by using the behavior modification services provides by other modules. In conventional programming, on the other hand, modules share information by directly manipulating shared data structures.

There is no attempt to hide the data structures of a module. By minimizing the number of modules that must understand how the data is structured one can minimize the number of modules that must be changed when the data structure changes.

Another feature of object-oriented programming is the notion of inheritance. Inheritance enables the easy creation of objects that are almost like other objects with a few incremental changes. When an object is declared as a specialization of another object it inherits the properties and behavior of the object. Inheritance relieves the programmer of the need to specify redundant information. It simplifies updating and modification, since the inherited information is entered and changed in only one place.

Many of the ideas behind object-oriented programming originated in SIMULA [Dahl 1966]. In SIMULA, objects are grouped into classes and classes themselves can be organized in a subclass hierarchy. Objects are similar to records with functions as components. A subclass inherits the attributes of its superclass. Because of inheritance, elements of class can appear anywhere an element of its superclass can appear. In SIMULA, objects are represented by coroutines, so that communication between objects is implemented as "message passing" between processes.

The first interactive, display-based implementation of the object-oriented paradigm was SMALLTALK [Goldberg 1983]. SMALLTALK adopted the inheritance concept of SIMULA, but its objects are not processes and message passing is implemented as a function call. Even though it is implemented as a form of indirect procedure invocation, SMALLTALK still emphasizes the message passing concept. Instead of naming a procedure to perform an operation on an object, a message is sent to the object. Objects respond to the message by using the procedures (called "methods") for performing the operation. In order to gain the flexibility of interactive use, SMALLTALK abandoned SIMULA's strong typing and static scoping.

In the case of both SIMULA and SMALLTALK inheritance is simple, i.e., the subclass hierarchy has the form of a tree. In simple inheritance a subclass inherits its attributes from just one class. Multiple inheritance, on the other hand, occurs when a class can be considered as a subclass of two or more incompatible classes. The subclass hierarchy is no more in the form of a tree, but a lattice.

Most of the languages that implement multiple inheritance have done so in the context of a type-free language, often implemented as extensions of LISP. GALILEO [Albano 1985] and OBJ [Goguen 1984] are the exceptions. In these languages multiple inheritance is realized within the framework of strong typing [Cardelli 1984].

Over the past few years the object-oriented paradigm has become very popular in the AI community, often as add-ons to LISP. The reason for its popularity is because it supports the structuring and organization of knowledge in the problem domain. The object-oriented paradigm provides the knowledge engineer with an easy means for describing the objects in the domain.

2.2 Towards the Amalgam

Each of the paradigms discussed in the previous section possesses important and useful constructs for developing knowledge-based systems. However, no single paradigm provides all the features necessary for knowledge engineering [Bobrow 1985]. They allow for different things to be stated concisely, and thus are suitable for different classes of problems.

For example, while the object-oriented paradigm supports the modeling of domain objects, it provides no facilities for representing domain knowledge that is in the form of decision making rules. Logic programming, on the other hand, provides facilities for the representation of rules, but its object modeling capabilities are minimal.

The choice of a paradigm for a particular problem is based on whether there is close fit of the paradigm to the problem at hand. If the appropriate paradigm is not chosen, elaborate and often obscure techniques have to be used to express the problem.

Several researchers have argued the need for incorporating many paradigms within the same environment as a means to providing all the constructs necessary for building knowledge-based systems [Bobrow 1985], [Fikes 1985]. They have tried to integrate the paradigms by embedding one or more paradigms on top of another. Numerous multiparadigm environments for developing knowledge-based systems have been built; the most notable ones being LOOPS [Bobrow 1983] and KEE [Fikes 1985].

While we share their view of the need to combine the paradigms, we take a different approach to the problem. We start by identifying the primitive language features from the various paradigms that are necessary for building knowledge-based systems. We then introduce these features in their minimal and essential form in a new language. Our goal was to study the semantics of combining the paradigms.

Another important difference between our approach and that of the others is that we wanted to explore the safe blending of language features in a strongly typed language.

2.3 Multiparadigm Environments

There are many commercially available multiparadigm environments for building knowledge-based systems. In this section we present a brief overview of the two most prominent ones — KEE [Fikes 1985] and LOOPS [Bobrow 1983]. We conclude the section by examining the limitations of these multiparadigm environments.

The first system, KEE, a product from Intellicorp, is a hybrid knowledge engineering tool that combines frame-based knowledge representation, rule-based reasoning, LISP functions, interactive graphics, and active values. Object-oriented programming provides a unifying principle for these different methodologies. The KEE system has been applied to a variety of problems.

In the KEE system the user describes the domain by declaring classes and instance of these classes. Each instance or class is represented by a frame. Frames can be organized into taxonomies using two constructs that represent the relationship between frames — member links, representing class membership, and subclass links, representing class containment.

An important source of the expressive power of KEE is the extensive facility that is provided for describing object attributes. Although KEE provides no specific facilities for declaring behavioral knowledge, it provides various ways of attaching procedural information to frames. This procedural attachment provides a form of object-oriented programming whereby objects represented by frames can respond to messages.

Rules in KEE are very simple in form. The antecedent is a simple conjunction of predicates on the attribute values of objects, or on the membership of objects in specific classes. The consequent fills in the values of one or more attributes. There is no variable binding or pattern matching during inference. The default strategy for control during inference is backward chaining, but the user can modify this control strategy.

LOOPS is a multiparadigm programming environment for knowledge engineering implemented in Interlisp-D. While it is available commercially, it has primarily been used as a research vehicle for experimenting with the various knowledge representation mechanisms. LOOPS takes full advantage of the rich environment provided by Interlisp-D.

LOOPS combines procedure-oriented programming (INTERLISP), object-oriented programming, and access-oriented (e.g., demons and attached procedures) with rule-based programming. The tool was developed to aid the design of expert systems. It was initially used as an expert assistant for the design of integrated digital systems.

Rules in LOOPS are organized into modular components called *rule sets*. Rule sets consist of an ordered set of rules and a control structure. A rule's antecedent consists of a list of LISP expressions that are evaluated from left to right. Similarly the rule's consequent is a list of expressions that are evaluated.

Unlike other systems, LOOPS has explicit control structures for its rules, i.e., there is no factoring out of the control from the domain knowledge. Rule sets in LOOPS may be invoked by sending a message to an object, or they can be invoked as a side effect of fetching or storing an active value. They can also be invoked directly from LISP as functions.

Programming environments, such as LOOPS and KEE, provide powerful primitives for programming in many paradigms. They achieve this by embedding one or more paradigms on top of another. The important issue, however, is not how many paradigms a particular system has, but whether the paradigms are cleanly and elegantly integrated. For the most part programming environments such as LOOPS and KEE have ignored the semantic issues of combining the paradigms.

One drawback of these environments is that, since they are built on top of LISP, they lack facilities for representing arbitrary relations among objects and reasoning based on these arbitrary relations. Unlike logic languages, rules in these environments are very simple in form. There is no concept of logic variables and the pattern matching facility that is used during inference is certainly not as powerful as unification.

Another drawback of these environments is that they are all type free. With type information the system can detect certain kinds of errors that conflict in an obvious manner with the domain.

2.4 PROLOG

PROLOG [Clocksin 1984] provides facilities for the declarative representation of knowledge and a powerful inference capability. Some of the features of PROLOG that make it attractive for knowledge engineering also render it unique among programming languages. They include:

- Powerful symbol manipulation facilities, including unification.
- Support for search-based computation and backward chaining.
- The invertibility of logic programs allows for the use of programs for more than one purpose.
- The simple and elegant semantics allows for a cohesive framework for building rule-based expert systems.

While PROLOG has several significant assets that make it suitable for building knowledge-based systems, it also has many drawbacks. In the logic interpretation of PROLOG, first-order terms which are not variables appear as Skolem constants or functions. These functions are never evaluated. Rather, they are used operationally as record constructors. Functional first-order terms in PROLOG behave as instantiated, partially instantiated, or uninstantiated record structures. In PROLOG, the only way to organize data is through the use of functional first-order terms.

For example, the term "complex_number(I,J)" can be viewed as an uninstantiated record containing two fields, the real and imaginary parts of a complex number. The functor (record constructor) "complex_number" is an integral part of every occurrence of a complex number in the PROLOG program.

A fundamental limitation of the use of terms to represent records in PROLOG is that the interpretation of the argument positions is not transparent to the user. For example, in the term "complex_number(2,3)" it is not clear whether the first argument represents the real or the imaginary part of the complex number. The burden of correctly interpreting the arguments falls on the programmer.

Another fundamental limitation of using terms to organize data is that there are no restrictions to using the same functor (record constructor) to represent two different record structures, each with a different number of components. Thus the careless omission of a component in a record goes undetected; the term is mistakenly assumed to be a record of a different type.

In PROLOG, since functional first-order terms are never evaluated, functional behavior has to be simulated. Rules and facts are used to simulate functional behavior. The relative ordering of the rules and facts is often critical in such a case. If this is not done the program may enter an infinite loop. In some situations cuts may have to be used to prevent unnecessary backtracking.

As a consequence, the "clean" control of program execution becomes a challenging problem in PROLOG — one which can be significantly alleviated if the functional paradigm were available. Furthermore, while functional behavior can certainly be simulated by using relations, it is somewhat artificial to view what is naturally thought of as a one-way function as a two-way relation. While the invertability of relations is certainly an asset in some cases, it is certainly not the best paradigm to use in all situations.

PROLOG's flat collection of rules and facts lacks facilities for constructing hierarchies of concepts. Inheritance can be captured by the semantics of logical implication. For example, to assert that "all men are mortal" in PROLOG we write:

```
mortal(X) :- person(X).
```

The above rule which would read formally as "for all X, X is mortal if X is a person", is semantically satisfactory. However, as Aït-Kaci and Nasr [Aït-Kaci 1986b] observed, inheritance captured in this manner leads to a lengthening of proofs. They propose a simple and efficient solution to the problem. We discuss in detail their solution in the next section.

PROLOG's power is derived from its ability to represent arbitrary relations among objects and deduction based on these relations. While this is a powerful feature, the lack of constraints on the type of objects that can participate in a relation often results in programs that are syntactically correct but semantically meaningless. For example, given that the relation "grandparent" is a relation involving two persons — the grandparent and grandchild — there is nothing that prevents the programmer from representing a fact where the predicate is "grandparent" but where the objects are of some other type than person.

A drawback of PROLOG's unification process is that it is a purely syntactic process. While we can match the terms $+(3, X)$ and $+(3, 4)$, we cannot match the terms $+(3, X)$ and $+(2, 5)$, even though we know that if X were bound to 4, the two terms would be semantically equivalent.

These drawbacks were the motivation behind the development of SEMLOG. SEMLOG provides significant gains over PROLOG:

- In SEMLOG, multiple inheritance is realized through a type-object lattice, not through the semantics of logical implication. Thus inheritance, which is a special kind of relation between objects, is separated from logical inference process.
- Unlike PROLOG, where functional first-order terms are never evaluated, SEMLOG provides true functions. Thus the programmer is provided with a mechanism to write program segments where the underlying computational model is based on functional evaluation.
- Records in SEMLOG are more powerful than their counterparts in PROLOG. In SEMLOG records can have functional components. Functional components in records are used to model *methods* of object-oriented programming. Field selection is used to model *message passing*.
- Because SEMLOG is a strongly typed language, certain kinds of programming errors which go undetected in PROLOG, are reported to the programmer prior to execution. This saves the programmer considerable time and effort during the development process.

2.5 Incorporating Inheritance in the Unification Process: LOGIN

The language LOGIN [Aït-Kaci 1986b] addresses some of the limitations of PROLOG. The language is an elaboration of the PROLOG language.

In LOGIN, PROLOG's first-order term is replaced by the more general ψ-term. This extended form of terms allows the incorporation of inheritance information directly in the unification process rather than indirectly through the semantics of logical implication. The result is that inheritance is performed more efficiently. Another advantage to separating inheritance from the resolution-based inference mechanism is that the resultant programs are more easily understood. In other words, the expressive power of the language is enhanced.

The generalized first-order terms of LOGIN are called ψ-terms. Unlike first-order terms in PROLOG where record fields are not labeled, the fields of a ψ-term are explicitly labeled by symbolic keywords. Clearly, the explicit labeling of the record fields by symbolic keywords is better than their implicit labeling based on position.

In LOGIN, taxonomic information is captured in a lattice structure, called the signature. This information is used by the ψ-term unification algorithm to realize inheritance. LOGINS's ψ-term unification algorithm is a modification of the method used by Huet for the unification of regular first-order terms based on a fast procedure for congruence closures. Huet's algorithm was devised for conventional fixed-arity terms with arguments identified by position, and over flat signatures. LOGIN's unification algorithm, on the other hand, is devised for the more general ψ-terms which are free from these restrictions. The algorithm uses the inheritance information in the signature to compute the *greatest lower bound* of two ψ-terms. The reader is referred to [Aït-Kaci 1986b] for a more detailed presentation of the algorithm.

LOGIN has successfully solved the inheritance problem by incorporating inheritance in the unification process. Compared to PROLOG, inheritance is more easily captured and more efficiently used in LOGIN. However, there are other forms of knowledge for which LOGIN still lacks adequate representational facilities.

LOGIN has no facilities for defining functions. Just like PROLOG, functional behavior can be simulated in LOGIN by using relations. But as we had pointed out earlier, the consequence of this is that the "clean" control of program execution becomes a challenging problem. The mixing of the explicit "cut" operator with the underlying implicit control strategy of the inference process often leads to subtleties in the execution of a program. These subtleties are not easy to discern from a superficial examination of the code. The relational formulation of functional behavior often leads to programs that are unnecessarily opaque.

While LOGIN's ψ-terms are certainly better than first-order terms for modeling domain entities, they are not as powerful as objects in the object-oriented paradigm. LOGIN's ψ-terms cannot have functional components.

SEMLOG provides significant gains over LOGIN in terms of knowledge representation:

- In contrast to LOGIN, SEMLOG provides functions. The programmer is provided with a direct mechanism to model functional behavior.
- Records in SEMLOG are more powerful than their counterparts in LOGIN — ψ-terms. They can have functional components, and thus they are better able to model domain objects than ψ-terms
- SEMLOG allows the arguments of its literals to be expressions. Expressions subsume variables, records, variants, functions, function applications, field selection, and case statements. By allowing the arguments of literals to be expressions we considerably enhance the expressive power of the language to model the domain. In the next section, after we introduce SEMLOG, we illustrate through an example this expressive power and compare it with LOGIN.

Object-Oriented Programming in PROLOG

There have been proposals that have attempted to provide a capability for object-oriented programming in PROLOG [Zaniolo 1984], [Stabler 1986], [Gullichsen 1985]. While the individual approaches differ from each other in implementation they are conceptually very similar. The object-oriented paradigm is realized in these systems through a collection of PROLOG clauses that implement the primitive features: (1) an object with an associated set of methods, (2) inheritance, and (3) message passing between objects.

Typically objects are represented in these systems with PROLOG facts of the form:

```
object(name,methods)
```

where *name* is any first-order term used to designate a particular object, possibly including parameters, and *methods* is a list of PROLOG rules for responding to messages. For example, the fact:

```
object(reg_polygon(No_of_sides, Length_of_sides),
       [ perimeter(P) :- P is No_of_sides*Length_of_sides,
         what_is_it('a regular polygon') ])
```

is an object *reg_polygon*, specified with two parameters. Methods associated with the object are represented by a list of PROLOG clauses.

The inheritance relation between objects is explicitly stated in these systems by asserting an *isa* relation. For example, if we want to store that a square with sides of length L is a *reg_polygon* with four sides of length L we do the following:

```
object(square(Length_of_side), [ what_is_it('a square') ]).

isa(square(L), reg_polygon(4, L).
```

With the above representation we should be able to compute the perimeter of the square using the inherited method that applies to all regular polygons.

The clauses that implement message passing in these systems will try the methods associated with the object to which the message is directed. If those fail to apply, it uses the *isa* relation to find the inherited methods.

Integrating logic and object-oriented paradigms has been the primary motivation for the development of these systems. Researchers building these systems claim that by providing object-oriented features on top of existing PROLOG systems, the much desired object-oriented metaphor is made available while still retaining the advantage of logic programming. What is not clear, however, is the underlying semantics of these formalisms.

In terms of knowledge representation, what these systems lack, besides clear semantics, are functions and type checking.

3 The SEMLOG Language

In this section we present the SEMLOG language. Section 3.1 is an introduction to the language. and the major phases of programming in the language are discussed. we discuss the major phases of programming in the language. Section 3.2 is a summary of the language features. The formal syntax of the language is also presented in Section 3.2 An example in the language is presented in Section 3.3. In Section 3.4 and 3.5 we compare the expressive power of the language to the two logic languages PROLOG and LOGIN by revisiting the example of Section 3.3. In Section 3.6 extend the example of Section 3.3. to illustrate the object-oriented features of the language. In Section 3.7 we show how the language lends itself to a problem from the decision support systems area.

3.1 Introduction

Our study of the three paradigms — object-oriented, logic, and functional programming — revealed that while each paradigm possessed important and useful features for knowledge representation, no single paradigm possessed all the features. Our goal was to develop a language in which the diverse forms of knowledge about a domain can be safely and concisely expressed.

We set out by first defining the requirements that would characterize such a knowledge representation language. The following are the requirements:

1. The language must provide primitives for modeling domain objects.
2. The language must support the notion of inheritance.
3. The language must allow for function definitions and function applications as a means of computing.
4. The language must provide for the representation of arbitrary relations among objects, and deduction based on these arbitrary relations.
5. The language must be strongly typed.

The programming language SEMLOG was developed with the above requirements in mind. The language is a strongly typed, multiparadigm, high-level language that was developed to satisfy the diverse needs of representing domain knowledge. The language provides powerful primitives for representing domain objects and their attributes, relations in which domain objects participate, and decision making rules. The language is the result of a synthesis of language features from three different paradigms: object-oriented, logic, and functional programming.

Because the language is strongly typed, these diverse forms of knowledge can be safely expressed. Through type checking the language ensures the detection of erroneous knowledge in the system. Erroneous knowledge is knowledge that conflicts in an obvious manner with the specification of the domain.

SEMLOG is a logic language in that it allows for programming with relations. However, unlike logic languages such as PROLOG, where the arguments of a relation are restricted to variables and first-order terms (PROLOG's objects), SEMLOG allows its predicate arguments to be expressions. Expressions in SEMLOG subsume variables, records, variants, function definitions, function applications, record field selection, and case statements.

As a result the language is more expressive than PROLOG. Domain knowledge can be stated concisely in the language. One does not have to resort to the use of extensive data structures and obscure techniques to capture domain knowledge.

Another difference between SEMLOG and PROLOG, is that in SEMLOG the participants of a relation are restricted to be from specific domains. In other words, the arguments of a relation are typed. The result is a well-typed knowledge base of objects, facts, and rules.

Multiple inheritance is realized in the language through the type system. There is an implicit inheritance relation between types that is based on the structure of the types. This is in contrast to object-oriented languages, such as SMALLTALK [Goldberg 1983] and FLAVORS [Weinreb 1981], where classes, the equivalent of types in these languages, are explicitly named and where the inheritance relation between classes is explicitly declared.

Programming in SEMLOG can be divided into three fundamentally distinct phases. The first phase involves the specification of the domain. The domain is specified by a series of type and relation declarations. Type declarations describe the various entity types in the domain. Relation declarations specify associations between entity types in the domain.

In the second phase the individual objects, facts and rules for a particular instance of the domain are entered. Domain objects are represented in the systems by creating instances of entity types. A domain object, once created, can be bound to an identifier; any subsequent reference to the identifier in the system is a reference to the entity that the identifier denotes. Facts state associations among objects in the domain. Rules are used to express facts that depend on other facts. All the domain objects, facts, and rules entered in this phase must be type consistent with the domain specifications of Phase 1. Through type checking the language ensures this type consistency.

In the third phase the user enters into an interactive dialogue with the system. The user enters either an expression or query into the system and the system responds with one or more solutions. The system's response to an expression is its value. If a query is entered, the system responds by finding the solutions that satisfy the query. As in Phase 2, the type consistency of the expressions and queries entered is checked against the domain specifications of Phase 1.

3.2 Language Features

In this section we present a detailed summary of the features of the language. We discuss types, expressions, multiple inheritance, predicates, facts, rules, and queries. Finally we present the formal syntax of the language.

3.2.1 Types

Types arise informally in any domain to categorize objects according to their usage and behavior. The classification of objects in terms of the purpose for which they are eventually used results in well-defined systems.

Base Types The language provides three base types — *bool*, *int*, and *string*. The reason for restricting the language to only three base types was motivated by a desire to keep the language manageable. Other base types can be added to the language without significantly changing the semantic issues.

Records Records are unordered, labeled sets of values: $[a := 3; b := true]$ has type $[a : int; b : bool]$, where a and b are labels. In general a record $[l_1 := e_1; \ldots; l_n := e_n]$ has type $[l_1 : \tau_1; \ldots; l_n : \tau_n]$, where $l_1, \ldots, l_n$ are labels, $e_1, \ldots, e_n$ are expressions, $\tau_1, \ldots, \tau_n$ are types, and where the expressions $e_1, \ldots, e_n$ are of types $\tau_1, \ldots, \tau_n$, respectively. Labels are a separate domain; they are not identifiers or strings, and they cannot be computed by an expression in the language.

Variants While records are labeled cartesian products, a variant is a labeled disjoint sum. A variant type looks very much like a record type; it is an unordered set of label-type pairs, enclosed in curly braces instead

of brackets. An element of a variant type is a labeled expression, where the label is one of the labels in the variant type, and the expression has a type matching the type associated with that label.

For example, an object of type $\{a : int; b : bool\}$ is either an integer labeled a or a boolean labeled b. Thus the two objects $\{a := 5\}$ and $\{b := false\}$ are objects of type $\{a : int; b : bool\}$.

Functions A function can be defined in the language using the lambda expression of the form: **fun** $(x : \tau)\,.\,e$, where x is the parameter of the function whose type is τ, and where e is an expression — given in terms of the parameter — that is used to compute the value of the function. Lambda expressions in the language evaluate to functions. The type of the function **fun** $(x : \tau)\,.\,e$ is $\tau \rightarrow \sigma$, where τ is the type of the argument, and σ the type of the result of function evaluation.

For the sake of simplicity we allow for only unary functions in the language, i.e., functions with only one argument. The language does not allow recursive type or function definitions.

Since functions in the language are first-class values, they may be passed as arguments to other functions or be the result of function evaluations. It is important to note that a function is treated like any other object in the system. Hence, records in the language can have functional components.

Type Declarations Type declarations in the language introduce names for type expressions. Names for type expressions serve as abbreviations; they do not create new types. In other words, we use *structural equivalence* on types, instead of *name equivalence*. Two types are equivalent in the language only when they have the same structure.

3.2.2 Expressions

Constants. The simplest form of expression in the language is the *constant*. Constants are elementary expressions whose values are defined by the language. Some examples of constants in the language: *true*, 434, *"string"*, $[a := 5; b := true]$, and $\{c := 455\}$.

Identifiers. An identifier in the language is a symbolic name that denotes a value. The association of an identifier to a value establishes a binding, and the binding is effective during the entire interactive session.

Logic Variables. Logic variables have meaning only in the context of facts, rules, and queries in the language. Hence, a description of them would be premature at this point.

Arithmetic Operators. Composite expression are formed in the language by using arithmetic operators, such as $+$, $-$, $*$, and $/$.

Field Selection. The only operation on records is field selection. Field selection is used to extract a component of the record. Field selection is denoted by $r.l$, where r is a record, and l is the label of the component of the record to be extracted.

Case Statement. Variants can be inspected in the language by the case statement. For example, in the case statement:

$$
\begin{array}{l}
\textbf{case}\ \ e\ \ \textbf{of} \\
\quad \{l_1 :: i_1\} \Rightarrow e_1; \\
\quad\quad \vdots \\
\quad \{l_j :: i_j\} \Rightarrow e_j; \\
\quad\quad \vdots \\
\quad \{l_n :: i_n\} \Rightarrow e_n \\
\textbf{endcase}
\end{array}
$$

the contents of the variant e is bound to i_j if the variant tag is l_j, and then e_j is evaluated. The scope of each i_j is the corresponding e_j.

Functions. Functions were discussed in detail in an earlier part of this section.

Function Application. The only operation on functions is function application: $f(a)$, where f is the function and a the argument. The function is evaluated first, then the argument, and finally the application is performed.

3.2.3 Multiple Inheritance

Multiple inheritance is realized in the language through the type system. The types in the language are related to one another by the *subtype* relation. We say that a type σ is a *subtype of* or is *included in* another type τ when all the values of type σ are also values of type τ. In other words type σ *inherits* the attributes of type τ. Inheritance may be viewed as an abbreviation mechanism that avoids redefining the attributes of type τ in the definition of type σ. Inheritance, however, is more than a shorthand notation. It imposes structure upon a collection of related types and thus greatly reduces the complexity of the system specification.

The subtype relation permits the flexible use of objects of one type for objects of another type. Wherever an object of type τ is permitted, the language allows for an object of type σ, if σ is a subtype of τ. The general notion of subtyping specializes to different subtyping rules for different type constructors.

For example, in the case of records, a record type τ is a subtype of record type τ', if τ has all the fields (attributes) of τ', and possibly more, and the types of the common attributes are, respectively, in the subtype relation. Intuitively, anywhere a record of type τ' can occur a record of type τ can also occur; all that can be done with a record is field selection. The following is the subtyping rule for record types, where "$\leq$" denotes subtype:

$$[l_1 : \sigma_1; \ldots; l_m : \sigma_m; \ldots; l_n : \sigma_n] \leq [l_1 : \tau_1; \ldots; l_m : \tau_m]$$
$$\text{iff} \quad \sigma_i \leq \tau_i \quad \text{for} \quad i \in 1 \ldots m$$

Subtyping on record types corresponds to the concept of subclassing (inheritance) in object-oriented languages.

In contrast to record types, a variant type τ is a subtype of another variant type τ', only if τ has all the fields of τ', or possibly fewer, and the types of the common attributes are, respectively, in the subtype relation. This is because, case statements which are used to decompose variants into a fixed number of possibilities, works on all variants with those possibilities and fewer. The subtyping rule for variant types is:

$$\{l_1 : \sigma_1; \ldots; l_m : \sigma_m\} \leq \{l_1 : \tau_1; \ldots; l_m : \tau_m; \ldots; l_n : \tau_n\}$$
$$\text{iff} \quad \sigma_i \leq \tau_i \quad \text{for} \quad i \in 1 \ldots m$$

For function spaces, we have the following subtyping rule:

$$\tau_i \rightarrow \sigma_i \leq \tau_j \rightarrow \sigma_j$$
$$\text{iff} \quad \tau_j \leq \tau_i \quad \text{and} \quad \sigma_i \leq \sigma_j$$

Notice the reversal in the domain type of the functions. This can be easily explained. A function f, of type $\tau_j \rightarrow \sigma_j$, can be applied to any argument of type τ_j. A function g, of type $\tau_i \rightarrow \sigma_i$, can be used in the place of f because any argument of type τ_j is in the domain of g, since τ_j is a subtype of τ_i.

The set of all types in the language when ordered by the subtyping relation form a lattice. The top of the lattice is the type *Top* ($\top$); it denotes the type of all the values in the language. The bottom of the lattice is denoted by *Bottom* ($\bot$) and it represents the inconsistent type, i.e., there are no values of this type. In Section 4.3 we discuss the type lattice in more detail.

The language discussed thus far is essentially identical to the language studied by Cardelli [Cardelli 1984].

3.2.4 Logic Programming

In the language, *facts* and *rules* are used to represent the interrelationships between objects. Given facts and rules, queries in the language are used to find unknown objects that satisfy one or more relations. Logic variables are used for this purpose. By invoking a query that contains logic variables one can solve for the variables. The result of a query is the substitutions for variables that satisfy the query.

Clauses. A *clause* in the language is an implication in which a conjunction of zero or more conditions implies a conclusion. The conclusion of a clause is separated from its conditions by a <=>. The conditions are separated from each other by a comma.

For example, the following clause:

$$A <=> B_1, B_2, \ldots, B_n$$

states that: that A is provable follows from B_i ($\forall i \in 1 \ldots n$) being provable. The B_i's are the conditions of the clause, and A the conclusion.

Clauses are either facts or rules. A fact is a clause that has no conditions. A rule, on the other hand, has one or more conditions.

Literals. The conclusion and conditions of a clause can be any *literal*. A literal in the language is an n-ary predicate of the following form:

$$p(e_1, \ldots, e_n)$$

where p is a predicate symbol, and the argument e_i's are valid expressions in the language. Although literals in the language appear to be very similar to their counterparts in PROLOG, there are important differences in their usage and interpretation.

In PROLOG, the arguments of a literal can be any valid PROLOG term. A valid PROLOG term is a *constant*, a *variable*, or an n-ary *function* applied to n terms. The functional first-order terms in PROLOG are never evaluated. They behave as instantiated, partially instantiated, or uninstantiated record structures.

In SEMLOG, however, the arguments of a literal can be any expression in the language. Expressions subsume logic variables, records, variants, functions, function applications, field selection, and case statements. By allowing the arguments of a literal to be expressions we considerably enhance the expressive power of the language.

Another difference between PROLOG and SEMLOG literals, is that in SEMLOG the arguments of a literal are restricted to be expressions from specific domains, i.e., the arguments of a literal are typed. The type of the arguments of a literal depends upon the *signature* of the predicate. This is different from PROLOG, where there are no restrictions on the arguments of a literal. Any valid term can occur as an argument to a literal.

Restricting the type of the arguments of a literal results in well-typed programs. Every literal in the program can be *statically* type-checked to verify that the type of the arguments do not conflict with the signature specification of the predicate. The execution of ill-typed programs is avoided.

A disadvantage of restricting the arguments of a literal to specific domains is a loss of flexibility. However, a rich type structure substantially eliminates this disadvantage.

The differences in interpretation between PROLOG and SEMLOG literals will become apparent in the next section when we discuss the semantics of the language.

Predicate Signatures. In the language the signature or type of every predicate must be specified prior to its use in a literal. For example, the following signature specification in the language:

$$\text{signature} \quad p(\alpha, [b : \beta])$$

states that the signature of the predicate p is $\alpha, [b : \beta]$, i.e., p is a relation (predicate) between expressions of type α and the record type $[b : \beta]$. Every occurrence of the predicate, p, in a relation must have exactly two arguments and their types must be α, and $[b : \beta]$ respectively.

Let o_1 and o_2 be objects of type α, o_3 and o_4 objects of type β, and o_5 and o_6 objects of type δ. The expression $[b := o_3]$ is a record of type $[b : \beta]$. The following is a valid literal in the language:

$$p(o_1, [b := o_3])$$

Let the identifier m be bound to record $[b := o_4]$ in the environment, and identifier n to the record $[d := o_5]$. Let f be a function whose type is $[b : \beta] \rightarrow \beta$, and let $\beta, [d : \delta]$ be the signature of the predicate q. Here are some additional examples of valid literals:

$$q(m.b, n)$$
$$q(f(m), [d := o_6])$$

The argument $m.b$ in the first literal is field selection, and $f(m)$ in the second literal is function application. The type of every expression in the language can be determined statically.

The expression $m.b$, n, and $f(m)$ in the above literals are not in normal form; they are shown in the form the user enters. The system, however, replaces the above expressions by their normal forms, when it stores the literals in the knowledge base.

Here are some examples of literals which have type errors:

$$q([d := o_6], f(m))$$
$$q([b := o_3], n)$$

The first literal is invalid because its arguments have been transposed. The order of the expressions in a literal is important. The second literal expects an expression of type β as its first argument; instead its first argument is of type $[b : \beta]$.

In SEMLOG, when we say that a type τ is a subtype of type σ, we are implying that whatever properties objects of type σ possess, objects of type τ must also possess. The potential to participate in a relation can be viewed as a property of the object.

Let r be a predicate whose signature is σ, η. Clearly any object of type σ can participate in a relation that involves the predicate r. Since type τ is a subtype of type σ, objects of type τ can also participate in relations that involve the predicate r.

For example, if s is a predicate whose signature is $[a : \alpha], \beta$, and g a function whose type is $[b : \beta] \rightarrow \beta$, the following is a valid literal:

$$s([a := o_1; b := o_3], g([a := o_1; b := o_3]))$$

In the above literal, the function g can be applied to the argument $[a := o_1; b := o_3]$, because the type of the argument is a subtype of the domain type of the function. The above literal is valid because, the type of the first argument is a subtype of $[a : \alpha]$, and the result of function, g, is of type β.

It is important to note, that in general in the language, anywhere an expression of type σ is allowed it can be replaced by an expression of type τ, if $\tau \leq \sigma$.

Facts and Rules We saw in the preceding section that the signature of every relation (predicate) is to be specified prior to its use in a literal. In the language there are two types of relations — *primitive* and *defined* relations.

Primitive relations are relations that denote associations among entity types in the domain. These associations are independent of other relations in the domain. Primitive relations are declared during the first phase of programming in the language. We declare a primitive relation by specifying its signature.

Once primitive relations have been declared, *facts* based on these primitive relations can be asserted. Facts state associations among individual entities, not among the types of the entities.

Consider the following predicate signatures:

$$\textbf{signature } q_1([b : \beta], [d : \delta])$$
$$\textbf{signature } q_2(\alpha, [b : \beta])$$

Let o_1 and o_2 be objects of type α, o_3 and o_4 objects of type β, and o_5 an object of type δ. Suppose we want to specify that the two objects $[b := o_3]$ and $[d := o_5]$ are related to each other by the relationship q_1. We do this in the language by asserting the following fact:

$$\textbf{fact } q_1([b := o_3], [d := o_5])$$

The arguments of the above fact are consistent with the signature of the predicate q_1 namely $[b : \beta], [d : \delta]$.

Suppose we want to specify that all objects of type $[b : \beta]$ in the domain are related to the object $[d := o_5]$ by the relationship q_1. We do this by asserting the following fact:

$$\begin{array}{ll} \textbf{let} & U : [b : \beta] \ \textbf{in} \\ & \textbf{fact} \ \ q_1(U, [d := o_5]) \end{array}$$

In the above fact, the logic variable U stands for the object, of type $[b : \beta]$, in the domain. The logic variable U can only be bound to objects of type $[b : \beta]$. This is different from PROLOG where a logic variable can be bound to any term in the language.

Every logic variable in the language is typed. The syntactic scope of a logic variable is limited to the fact or rule that follows the declaration. Logic variables in the language are denoted by upper case letters.

Suppose m is an identifier that is bound to the record $[a := o_1; d := o_5]$ in the environment. The following fact asserts that the field labeled a in the record, identified by m, is related to the object $[b := o_4]$ by the relationship q_2:

$$\textbf{fact} \ \ q_2(m.a, [b := o_4])$$

In the above fact the expression $m.a$ is not in normal form; it is shown in the form the user enters. The system, however, stores the fact with the expression $m.a$ replaced by its normal form:

$$\textbf{fact} \ \ q_2(o_1, [b := o_4])$$

Defined relations are relations that denote associations among entity types. But unlike primitive relations, these associations are dependent on other relations. In other words, the relation is defined in terms of conditions or constraints that, when met, implies that the relationship holds. Defined relations are more commonly referred to as *rules*.

To state rules in the language, we need logic variables. A logic variable stands for the same object wherever it occurs in the rule. Consider the following rule in the language:

$$\begin{array}{ll} \textbf{let} & P : [a : \alpha] \, , Q : [b : \beta] \ \textbf{in} \\ & \textbf{rule} \ \ q_4(P, Q.b) <=> q_1(Q, [d := o_5]), q_2(P.a, Q) \end{array}$$

where q_4 is the defined relation — the head of the rule. The relations q_1 and q_2 are the conditions that need to be satisfied for q_4 to hold — the body of the rule. In the rule, P and Q are logic variables whose types are $[a : \alpha]$ and $[b : \beta]$, respectively. The rule reads: as for all objects P of type $[a : \alpha]$ and all objects Q of type $[b : \beta]$, if $q_1(Q, [d := o_5])$ and $q_2(P.a, Q)$ then $q_4(P, Q.b)$.

It is important to note, that it is not necessary to specify the signature of the predicate q_4 prior to the rule definition. Since the signature of q_4 is primarily dependent on the types of the expressions P and $Q.b$, it can be easily determined. The signature of q_4 is $[a : \alpha], \beta$. The syntactic scope of the logic variables P and Q is limited to the rule following their declarations.

We can define any number of rules for the same defined relation, but their signatures must all be consistent. When we say consistent, we mean that once the signature of a defined relation has been fixed, the signature of the defined relation in all subsequent rules must either match exactly with the original signature, or be a subtype of it. The first rule for the defined relation fixes its signature. A signature $\alpha_1, \ldots, \alpha_n$ is a subtype of another signature $\beta_1, \ldots, \beta_n$ only if $\alpha_i \leq \beta_i$ for all $i \in 1 \ldots n$.

For example, the following is another rule for the relation q_4:

$$\begin{array}{ll} \textbf{let} & X : [a : \alpha; e : \sigma] \, , Z : [d : \delta] \ \textbf{in} \\ & \textbf{rule} \ \ q_4(X, o_4) <=> q_2(X.a, [b := o_4]), q_1([b := o_3], Z) \end{array}$$

The rule reads: for all objects of type $[a : \alpha; e : \sigma]$ if there exists an object of type $[d : \delta]$ such that $q_2(X.a, [b := o_4])$ and $q_1([b := o_3], Z)$, then $q_4(X, o_4)$. A logic variable that appears in the head of a rule is universally quantified over objects in its domain. Logic variables that occur in the body, but not in the head, are existentially quantified over objects in their respective domains.

Queries. Given facts and rules in the language, *queries* in the language are used to find unknown objects that satisfy one or more relations. Logic variables are once again used for this purpose. By invoking a query that contains logic variables one can solve for the variables. The result of a query is the substitutions for the variables that satisfy the query. The following is a query in the language:

$$\begin{array}{ll} \textbf{let} & M : [a : \alpha], N : \beta \ \textbf{in} \\ & \textbf{list}\ M\ \textbf{such that}\ q_4(M, N) \end{array}$$

The above query finds the substitutions for the logic variable M that satisfy the relationship $q_4(M, N)$. The result of the query is a list of substitutions for the logic variable M. The literal $q_4(M, N)$ is known as the goal of the query.

A query can have more than one goal. If it has more than one goal, then all the goals have to be satisfied for the query to be satisfied. The system uses the known clauses—facts and rules—to satisfy the goals of a query.

3.2.5 Syntax

In the previous section we introduced the features of the language. In this section we summarize the syntax of the language. The following notational conventions are used throughout the syntax:

1. Keywords are in bold.
2. For any syntax class S, we define $S\_\text{lst} ::= S_1, \ldots, S_n$.
3. For any syntax class T, we define $T\_\text{seq} ::= T_1; \ldots; T_n$.
4. Superscripts are used on multiple usage of the same syntax class.

Figure 1 is the syntax for type declarations, Figure 2 the syntax for expressions, and Figure 3 illustrates the syntax for predicate signatures, facts, rules, and queries.

```
type-binding ::=
      type ident <=> type
type::=
      ident
      bool
      int
      string
      [ field-type_seq ]
      { field-type_seq }
      type –> type'
field-type ::=
      label : type
```

Figure 1: Type Declarations

3.3 An Example

In order to give the reader a flavor of the language we present an example taken from academic life. Figure 4 illustrates the conceptual model of a facet of university life. The ovals in the figure represent entity types, the solid lines denote inheritance between entity types, and dotted lines represent interrelationships between entity types.

```
ident-binding ::=
        val ident <=> expr
expr ::=
        boolean
        integer
        string-constant
        ident
        object
        logic-var
        expr.label
        case expr of discrim_seq endcase
        expr ( expr )
        fun ( ident : type ) . expr
        expr : type
discrim ::=
        { label :: ident } => expr
object ::=
        [ field_seq ]
        { field }
field ::=
        label := expr
```

Figure 2: Expressions

```
predicate-signature ::=
        signature ident ( type_lst )
fact-defn ::=
        fact literal
        let logic-var-decl_seq in fact literal
rule-defn ::=
        let logic-var-decl_seq in rule literal <=> prop_lst
query ::=
        let logic-var-decl_seq in list expr such that prop_lst
logic-var-decl ::=
        logic-var : type
prop ::=
        literal
        condition
literal ::=
        ident ( expr_lst )
condition ::=
        expr = expr
        expr != expr
```

Figure 3: Signature, Facts, Rules, and Queries

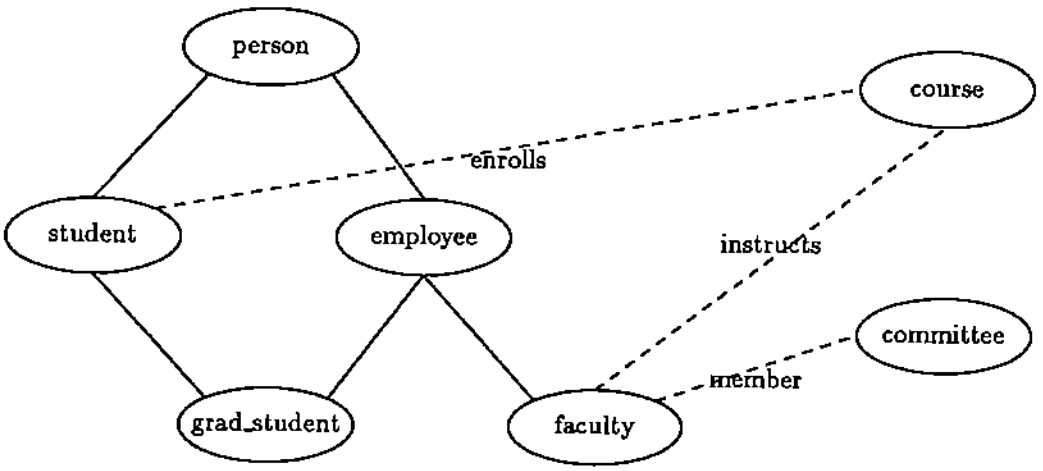

Figure 4: Conceptual model of university life

In the first phase of programming in the language we specify the conceptual model. We begin by describing the various entity types in the domain.

```
type person       <=> [name : string;
                       id : int]
type student      <=> [name : string;
                       id : int;
                       gpa : int]
type employee     <=> [name : string;
                       id : int;
                       dept : string]
type faculty      <=> [name : string;
                       id : int;
                       dept : string;
                       rank : sting]
type grad_student <=> [name : string;
                       id : int;
                       gpa : int;
                       dept : string;
                       adviser : faculty]

type course       <=> [dept : string;
                       number : int]

type committee    <=> [name : string;
                       dept : string;
                       meeting_room : int]
```

The entity type **faculty** is almost like the entity type **employee**. It has all the attributes of the entity type **employee**. Although we do not do so here, it is possible to develop a more convenient syntax that avoids redefining all the attributes of the type **employee** in the definition of type **faculty**. For example the following:

```
type employee <=> person and [dept : string]
```

states that the type **faculty** *inherits* all the attributes of type **employee**.
Next, we specify the signature of the primitive relations in the domain:

```
signature instructs(faculty, course)
signature enrolls(student, course)
signature member(faculty, committee)
```

The signature of a relation is a statement of the type of its participants. The order of the participants is important. For example, in the **enrolls** relation, the first argument is of type **student**, and the second argument of type **course**.

In the second phase of programming the individual objects, facts and rules for a particular instance of a domain are entered. We make the following definition of individual objects for the model of university life.

```
val smith <=> [name := ''smith'';
               id := 13612;
               dept := ''chemistry'']
val john  <=> [name := ''john'';
               id := 86448;
               gpa := 5]
val peter <=> [name := ''peter'';
               id := 28655;
               dept := ''mgmt'';
               rank := ''prof'']
val nancy <=> [name := ''nancy'';
               id := 43542;
               dept := ''cs'';
               rank := ''asst_prof'']
val tim   <=> [name := ''tim'';
               id := 24548;
               gpa := 6;
               dept := ''mgmt'';
               adviser := peter]

val mgmt600 <=> [dept := ''mgmt'';
                 number := 600]
val cs565   <=> [dept := ''cs'';
                 number := 565]
val cs502   <=> [dept := ''cs'';
                 number := 502]

val mgmt_grad_cmte    <=> [name :=
                             ''graduate committee'';
                           dept := ''mgmt'';
                           meeting_room := 402]
val cs_undergrad_cmte <=> [name :=
                             ''undergraduate committee'';
                           dept := ''cs'';
                           meeting_room := 210]
```

All the objects in this example happen to be records. None of them have functional components. However, SEMLOG does allow for records to possess functional components. In Section 3.6 we will extend the example presented here to illustrate this aspect of SEMLOG.

Relationships between objects are specified by asserting facts. For our example, we assert the following facts:

```
fact instructs(peter, mgmt600)
fact instructs(nancy, cs565)
```

```
fact instructs(nancy, cs502)

fact enrolls(john, cs565)
fact enrolls(john, cs502)
fact enrolls(tim, mgmt600)
fact enrolls(tim, cs565)

fact member(peter, mgmt_grad_cmte)
fact member(nancy, cs_undergrad_cmte)
```

A rule defines a new relation in terms of existing relations. For instance, to express the following statement:

> A faculty teaches a student if the student is enrolled in a course that the faculty member instructs.

we define the rule:

```
let F : faculty; S : student; C : course
in
  rule teaches(F, S) <=> instructs(F, C), enrolls(S, C)
```

The signature of the relation **teaches** is **faculty,student**.

Finally in the third phase the user engages in an interactive dialogue. The user gives a query and the system responds with all the solutions. For example, the query to find all the students who are taught by the faculty member **nancy** is stated as follows:

```
let STUD : student in
  list STUD such that teaches (nancy, STUD)
```

The system responds with a list of the students taught by the faculty **nancy** — the student **john** and the graduate student **tim**. The graduate student **tim** appears in the response because due to inheritance a graduate student is also a student.

Now, if we wanted only the graduate students who are taught by **nancy**, we would write:

```
let GS : grad_student in
  list GS such that teaches(nancy, GS)
```

The query

```
let S : student in
  list S.name such that teaches(tim.adviser, S)
```

finds the names of all the students who are taught by **tim**'s adviser. We use field selection to denote **tim**'s adviser. In SEMLOG, the arguments of a literal can be any expression. There is only one restriction. The type of expression must match the type that is specified by the signature of the relation.

Here are some additional queries. The query

```
let GS : grad_student in
  list GS.name such that teaches(GS.adviser, GS)
```

finds the names of all the graduate students who are taught by their adviser, i.e., they are enrolled in a course that their adviser instructs.

The query

```
let FAC : faculty; GS : grad_student in
  list GS.name such that teaches(FAC, GS),
                         FAC.rank = ''asst_prof''
```

finds the names of all the graduate students who are taught by assistant professors.

The query

```
let CMTE : committee in
  list CMTE.name such that member(tim.adviser, CMTE)
```

finds the names of all the committees that tim's adviser is a member of.

3.4 Comparison to PROLOG

The PROLOG representation of the facts and rules in the university example is shown below:

```
instructs(faculty(peter,28655,mgmt,prof), course(mgmt,600)).
instructs(faculty(nancy,43542,cs,asst_prof), course(cs,565)).
instructs(faculty(nancy,43542,cs,asst_prof), course(cs,502)).

enrolls(student(john,86448,5), course(cs,565)).
enrolls(student(john,86448,5), course(cs,502)).
enrolls(grad_student(tim,24548,6,mgmt,
                     faculty(peter,28655,mgmt,prof)),
        course(mgmt,600)).
enrolls(grad_student(tim,24548,6,mgmt,
                     faculty(peter,28655,mgmt,prof)),
        course(cs,565)).

member(faculty(peter,28655,mgmt,prof),
       committee(graduate_committee,mgmt,402)).
member(faculty(nancy,43542,cs,asst_prof),
       committee(undergradute_committee,cs,210)).

teaches(FAC, S) :- instructs(FAC, COURSE),
                   enrolls(S, COURSE).
```

The query to determine all the students who are taught by the faculty member **nancy** is stated as follows:

```
?- teaches(faculty(nancy,_,_,_), S).
```

PROLOG finds the solutions to this query, namely:

```
S = student(john,86448,5)
S = student(john,86448,5)
S = grad_student(tim,24548,6,mgmt,
                 faculty(peter,28655,mgmt,prof))
```

The term "student(john,86448,5)" occurs twice in the solution, because the student **john** is enrolled in both courses taught by the faculty **nancy**.

We run into difficulty when we try to formulate the query to find the names of all the students (including graduate students) who are taught by **nancy**. The query:

```
?- teaches(faculty(nancy,_,_,_), student(X,_,_)).
```

fails to find the names of the graduate students. The problem arises because we have not captured the inheritance relation between graduate students and students. We could have the following rule:

```
is_student(student(NAME,ID,GPA)) :-
            graduate_student(grad_student(NAME,ID,GPA,_,_)).
```

which states that "every graduate student is a student". But even this does not solve the problem. We leave it as an exercise to the reader to try formulating the query with the above rule.

The following rule, however, does the trick:

```
enrolls(student(NAME,ID,GPA), COURSE) :-
        enrolls(grad_student(NAME,ID,GPA,_,_), COURSE).
```

Aït-Kaci and Nasr observed that inheritance captured in this manner, i.e., through the use of logical implication, leads to a lengthening of proofs. They proposed a simple and efficient solution to the problem in the language LOGIN [Aït-Kaci 1986b]. We compare our work with theirs in the next section.

The query to determine the names of all the graduate students who are taught by their adviser can be expressed as follows:

```
?- teaches(FAC, grad_student(NAME,_,_,_,_,FAC))
```

But we run into difficulty in trying to determine the names of the graduate students who are taught by tim's adviser. To understand the difficulty, consider the following query:

```
?- teaches(FAC, grad_student(NAME,_,_,_,_,_))
```

In the above query we need to bind the variable, FAC, to tim's adviser. To do this we need to determine some goal in which tim participates, and which we know will always be true. Since, we know that tim is enrolled in a course, we could do the following:

```
?- enrolls(grad_student(tim,_,_,_,_,FAC), _),
    teaches(FAC, grad_student(NAME,_,_,_,_,_))
```

The above solution works because we know that there is a fact in which tim participates. But, this may not necessarily always be true. The important point to understand here is that the lack of expressions forces us to resort to obscure means to formulate the query. This is a fundamental limitation of PROLOG, one that has been effectively overcome in SEMLOG.

3.5 Comparison to LOGIN

LOGIN is an elaboration of the programming language PROLOG, in which the more general record like structure, known as the ψ-term, replaces the notion of first-order term [Aït-Kaci 1986b]. This extended form of terms allows for the integration of inheritance directly in the unification process rather than indirectly through the use of logical implication. The result is that inheritance is realized more efficiently in LOGIN than in PROLOG.

Although LOGIN provides an efficient methodology for capturing taxonomic information, it still has some drawbacks in representing other forms of domain knowledge, specifically expressions and functions. Through an example we illustrate the problem.

The LOGIN representation of the signature (not to be confused with the signature of relations in SEMLOG) for the example is as follows:

```
person = (name => string;
          id => integer).

student = person(gpa => integer).

employee = person(dept => string).

faculty = employee(rank => string).

gs < employee
gs < student
```

```
grad_stud = gs(adviser => faculty).

course = (dept => string;
          number => integer).

committee = (name => string;
             dept => string;
             meeting_room : integer).

smith = employee(name => 'smith';
                 id => 13612;
                 dept => 'chemistry').

john = student(name => 'john';
               id => 86448;
               gpa => 5).

peter = faculty(name => 'peter';
                id => 28665;
                dept => 'mgmt';
                rank => 'prof').

nancy = faculty(name => 'nancy';
                id => 43542;
                dept => 'cs';
                rank => 'asst_prof').

tim = grad_stud(name => 'tim';
                id => 24548;
                gpa => 6;
                dept => 'mgmt';
                adviser => peter(name => 'peter';
                                 id => 28665;
                                 dept => 'mgmt';
                                 rank => 'prof')).

mgmt600 = course (dept => 'mgmt';
                  number => 600).

cs565 = course (dept => 'cs';
                number => 565).

cs502 = course(dept => 'cs';
               number => 502).

mg_g_cmte = committee(name =>
                          'graduate committee';
                      dept => 'mgmt';
                      meeting_room => 402).

cs_u_cmte = committee(name =>
```

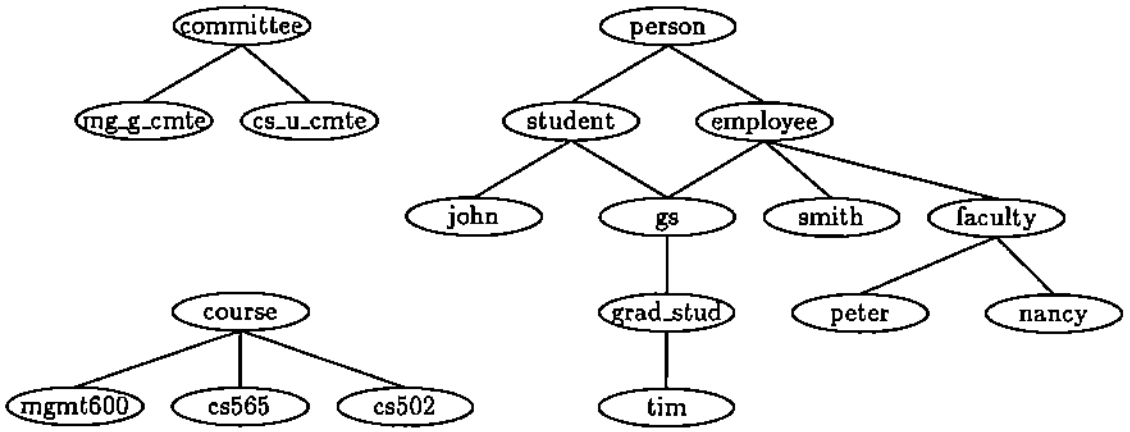

Figure 5: LOGIN signature

```
        'undergraduate committee';
dept => 'cs';
meeting_room => 210).
```

Given the above signature information, LOGIN constructs the signature for the example. Figure 5 is the constructed signature.

Now, we can define facts and rules in the context of this signature. For our example we have:

```
instructs(peter, mgmt600)
instructs(nancy, cs565)
instructs(nancy, cs502)

enrolls(john, cs565)
enrolls(john, cs502)
enrolls(tim, mgmt600)
enrolls(tim, cs565)

member(peter, mg_g_cmte)
member(nancy, cs_u_cmte)

teaches(F : faculty, S :student) :- instructs(F, C : course),
                                    enrolls(S, C).
```

The query to determine all the students who are taught by the faculty member, nancy, can be stated in LOGIN as follows:

```
?- teaches(FAC : nancy, STUD : student)
```

LOGIN's finds the following solutions:

```
STUD = john(name => 'john';
            id => 86448;
            gpa => 5)

STUD = john(name => 'john';
            id => 86448;
```

```
                gpa => 5)

STUD = tim(name => 'tim';
           id => 24548;
           gpa => 6)
           dept => 'mgmt';
           adviser => peter(name => 'peter';
                            id => 28665;
                            dept => 'mgmt';
                            rank => 'prof'))
```

If we were interested only in the graduate students who are taught by faculty member, **nancy**, we can coerce the type of the *tag* (variables in LOGIN) **STUD** in the above query to **grad_student**:

```
?- teaches(FAC : nancy, STUD : grad_stud)
```

The query to determine the names of all the graduate students who are taught by their adviser can be expressed as follows:

```
?- teaches(FAC, STUD : grad_stud(name => X; adviser => FAC))
```

In the above query, X is the answer we are seeking. But, we encounter the same difficulty that we countered in PROLOG, when we try to formulate a query determine the names of the students who are taught by **tim**'s adviser. The difficulty arises because in the following query:

```
?- teaches(FAC, STUD : student(name => X))
```

the *tag* (variables in LOGIN) FAC needs to be bound to **tim**'s adviser, the following ψ-term:

```
peter(name => 'peter';
      id => 28665;
      dept => 'mgmt';
      rank => 'prof')
```

To do this we need to find some goal in which **tim** participates, and which we know will always be true. As in PROLOG, we could do the following, because we know that the **tim** is enrolled in a course:

```
?- teaches(FAC, STUD : student(name => X)),
     enrolls(grad_stud(name => 'tim'; adviser => FAC), Z).
```

But, as before, this may not necessarily always be true. Once again the language feature that is missing is expressions, specifically in this case field selection.

The problem becomes more complicated if we are to the find the names of all the students who are taught by a faculty member who advises one or more graduate students. We could formulate the query as follows:

```
?- teaches(FAC, STUD : student(name => X)),
     enrolls(grad_stud(adviser => FAC), Z).
```

The above query will only work if every graduate student is enrolled in at least one course. If there happens to be a graduate student, who is his adviser's only student, and if he is not enrolled in any course, then we fail to find the students who are taught by his adviser.

In SEMLOG, we do not encounter these problems, because:

- Objects exists in the system independent of their participation in relations.
- The arguments of a literal in the language can be any expression. The only restriction being that the type of the expression not violate the signature of the predicate.

Expressions in SEMLOG subsume variables, records, variant records, function application, field selection, and case statements. Thus by allowing the arguments of a literal to be expressions we considerably enhance the expressive power of the language.

3.6 Extending the Example to Illustrate Object-Oriented Features

In SEMLOG, record types are used to model *classes*, and records are used to model class *instances*. The functional components in records are used to model *methods* and field selection is used to model *message passing*. Sending a message to an object with some parameters is simulated in SEMLOG through the selection of a functional component of a record and its application to the parameters.

Records are constructed explicitly by specifying at creation time the values of the components. There is no *create new instance of class X* operator in SEMLOG. At the time of the creation the functional components are bound to the individual records not to record types. Hence different records of the same record type can have different functional components. This is different from SMALLTALK and other object-oriented languages where the methods are bound to classes and shared by all the instances.

In this section we extend the example presented in Section 3.3 to illustrate the object-oriented features of the SEMLOG. We modify the type declarations of the entity types — employee, faculty, and grad_student — to include a functional component "pay". Thus we have:

```
type employee <=> [name : string;
                   id : int;
                   dept : string;
                   pay : int -> int]
type faculty <=> [name : string;
                  id : int;
                  dept : string;
                  pay : int -> int;
                  rank : string]
type grad_student <=> [name : string;
                       id : int;
                       gpa : int;
                       dept : string;
                       pay : int -> int;
                       adviser : faculty]
```

Some of the object bindings are changed to reflect the change in the type declarations:

```
val smith <=> [name := ''smith'';
               id := 13612;
               dept := ''chemistry'';
               pay := fun hrs. hrs * 15]
val peter <=> [name := ''peter'';
               id := 28655;
               dept := ''mgmt'';
               pay := fun hrs. hrs * 30;
               rank := ''prof'']
val nancy <=> [name := ''nancy'';
               id := 43542;
               dept := ''cs'';
               pay := fun hrs. hrs * 25;
               rank := ''asst_prof'']
val tim <=> [name := ''tim'';
             id := 24548;
             gpa := 6;
             dept := ''mgmt'';
             pay := fun hrs. hrs * 6;
             adviser := peter]
```

The functional component, pay, computes the weekly pay for an employee (faculty and graduate students included) based on the hourly wage and the number of hours worked in the week.

The signature of the primitive relations, the facts, and rules remain the same.
Now, the query

```
let FAC : faculty; GS : grad_student in
  list GS.name such that teaches(FAC, GS),
                         FAC.pay(40) = 1000
```

finds the name of all the graduate students who are taught by faculty whose weekly earnings are 1000. Notice, how message passing is accomplished by field selection and function application.

3.7 An Example from Decision Support Systems

The language SEMLOG lends itself very well to a problem from the decision support systems area. The interested reader is referred to Chapter 14 of "Foundations of Decision Support Systems" by Bonczek, Holsapple, and Whinston for a detailed presentation of the problem [Bonczek 1981]. We are primarily interested in the problem because it illustrates the usefulness of integrating the functional and logic paradigms.

Let us assume that we define functions to perform regression and prediction in SEMLOG. The regression function generates the regression coefficient and the prediction function when passed the regression coefficient and an independent variable observation generates the dependent variable observation:

```
regress <=> fun (independent-variable-observation : real;
                 dependent-variable-observation : real) ...
predict <=> fun (regression-coefft : real;
                 independent-variable-observation : real) ...
```

Although SEMLOG in its present form does not allow for functions to have more than one argument, we can easily extend the language to incorporate this facility.

Let us assume that we have a function to compute profit from revenue and expenses, and a function to compute dividends from profits and shares.

```
div-from-profit-and-shares <=> fun (profit : real;
                                    shares : int) ...
profit-from-revenue-and-expense <=> fun (revenue : real;
                                         expense : real) ...
```

Let the following facts represent the sales indicators, sales, expense indicators, and expense for the year specified:

```
fact sales-indicator-in-year(25555.62, 1988)
fact sales-indicator-in-year(34555.78, 1989)
fact sales-in-year(23345.77, 1988)

fact expense-indicator-in-year(21222.50, 1988)
fact expense-indicator-in-year(28777.62, 1989)
fact expense-in-year(21566.22, 1988)
```

Let us represent the following facts to relate the input and output parameters of the functions that we had defined earlier:

```
let SALES-INDICATOR : real; SALES : real in
  fact regression(SALES-INDICATOR, SALES,
         regress(SALES-INDICATOR, SALES))
```

```
let REGRESS-COEFFT :real, SALES-INDICATOR : real in
  fact prediction(REGRESS-COEFFT, SALES-INDICATOR
         predict(REGRESS-COEFFT, SALES-INDICATOR))

let EXPENSE : real, REVENUE : real, PROFIT : real in
  fact profit(REVENUE, EXPENSE,
         profit-from-revenue-and-expense(REVENUE, EXPENSE))

let PROFIT : real, SHARE : int, DIVIDEND : real in
  fact dividend(PROFIT, SHARES,
         div-from-profit-and-shares(PROFIT, SHARES)
```

Now, the rule that projects the revenue for a certain year:

```
let
  REVENUE : real, YEAR : int,
  SOME-OTHER-YR : int, SI-PREV : real,
  SI-CURR : real, SALES : real, REGRESS-COEFFT : real in
    rule revenue-in-year(REVENUE, YEAR) <=>
           sales-indicator-in-year(SI-CURR,YEAR),
           sales-indicator-in-year(SI-PREV, SOME-OTHER-YR),
           sales-in-year(SALES, SOME-OTHER-YR),
           regression(SI-PREV, SALES, REGRESS-COEFFT),
           prediction(REGRESS-COEFFT, SI-CURR, REVENUE)
```

The above rule states: if **SI-CURR** is the sales indicator for year **YEAR**, and if **SI-PREV** and **SALES** are the sales indicator and sales for some other year, then when **SI-PREV** and **SALES** are inputs to the regression function it outputs the regression coefficient **REGRESS-COEFFT**. When the regression coefficient along with the sales indicator for year **YEAR** are passed as input to the prediction function it outputs the revenue for the year **YEAR**.

Similarly, we can define rules for projecting the expenses, profit, and dividend for a given year:

```
let EXPENSE : real, YEAR : int,
    SOME-OTHER-YR : int, EI-PREV : real,
    EI-CURR : real, EXPENSE : real, REGRESS-COEFFT : real in
      rule expense-in-year(EXPENSE, YEAR) <=>
             expense-indicator-in-year(EI-CURR,YEAR),
             expense-indicator-in-year(EI-PREV, SOME-OTHER-YR),
             expense-in-year(EXPENSE, SOME-OTHER-YR),
             regression(EI-PREV, EXPENSE, REGRESS-COEFFT),
             prediction(REGRESS-COEFFT, EI-CURR, EXPENSE)


let PROFIT : real, YEAR : int,
    EXPENSE : real, REVENUE : real in
      rule profit-in-year(PROFIT, YEAR) <=>
             expense-in-year(EXPENSE, YEAR),
             revenue-in-year(REVENUE, YEAR),
             profit(REVENUE, EXPENSE)

let DIVIDEND : real, YEAR : int,
    PROFIT : real, SHARES : int in
      rule dividend-in-year(DIVIDEND, SHARES, YEAR) <=>
             profit-in-year(PROFIT, YEAR),
```

```
dividend(PROFIT, SHARES, DIVIDEND)
```

Given the above functions, facts, and rules, the following query in SEMLOG:

```
list DIVIDEND such that dividend-in-year(DIVIDEND, 100, 1989)
```

predicts the 1989 dividend, assuming that there are 100 outstanding shares.

4 The Semantics of SEMLOG

In this section we present a selective semantics of the language. Section 4.1 is an informal presentation of the resolution process. Section 4.2 is an introduction to the heart of the language—the semantic unification process. In Section 4.3 we present the data structure necessary to maintain the universe of objects — the type-object lattice. We discuss the significance of the type-object lattice with regard to the semantic unification process and present the necessary algorithms for maintaining the type-object lattice. In Section 4.4 the algorithm for semantic unification is presented. The resolution process in the language is illustrated through an example in Section 4.5. An example in the language involving subtypes is presented in Section 4.6.

We present the semantics of SEMLOG using ML [Milner 1984] which gives a clear means of expression the semantics an implementation at the same time.

4.1 Resolution

In SEMLOG, resolution is the rule of inference that is used to solve queries. The specific resolution strategy that is used in the language is a form of linear input resolution called *SL resolution*. This is the same strategy used by PROLOG. The interested reader is referred to [Clocksin 1984] for a detailed description of resolution. In this section we present a brief description of the resolution strategy in the context of SEMLOG.

The facts and rules that are entered into the SEMLOG system are the known clauses of the problem domain. A query in the language consists of a conjunction of goals to be satisfied. We start with the leftmost goal in the query and resolve it with one of the known clauses in the system to generate a new conjunction of goals. Then we resolve the leftmost goal of the new conjunction of goals to obtain another conjunction of goals, and so on. We continue this process until we are left with no more goals to be resolved, or until there are no clauses in the system that can be resolved with the chosen goal. If we are left with no more goals to be resolved then the query is satisfied, otherwise we have failed.

At each step, the clause that is chosen for resolution with the goal must satisfy the matching criteria — the head of the clause must match the goal under consideration. Once resolved, the goal is removed from the conjunction of goals and in its place the subgoals that make up the body of the matched clause are inserted. In other words, the subgoals that make up the body of the clause are appended to the *front* of the conjunction of subgoals. This means that SEMLOG finishes satisfying the newly added subgoals before it goes on to try something else.

If the clause that is chosen for resolution with the goal is a fact, no subgoals are added, since a fact has no body. A fact causes the goal to be satisfied immediately. A rule, on the other hand, reduces the task of satisfying the goal to the satisfaction of a conjunction of subgoals that make up the body of the rule. For example, if we resolve the conjunction of goals:

$$q_4(M, N), q_5(M.a, O)$$

with the rule:

$$\begin{array}{l}\textbf{let } P : [a : \alpha], Q : [b : \beta] \textbf{ in} \\ \quad \textbf{rule } q_4(P, Q.b) <=> q_1(Q, [d := o_5]), q_2(P.a, Q)\end{array}$$

we get the goals:

$$q_1([b := o_3], [d := o_5]), q_2(P.a, [b := o_3]), q_5(P.a, O)$$

The matching of the two literals—the goal, $q_4(M, N)$, and the head of the rule, $q_4(P, Q.b)$—is discussed in more detail in the next section.

If there is more than one clause that satisfies the matching criteria, SEMLOG considers one alternative at a time, fully exploring the alternative under the assumption that the choice is correct. The other alternatives are considered only after the chosen alternative has been fully explored. The clauses are considered in the order in which they are entered into the system.

While SEMLOG's resolution strategy is identical to the one used by PROLOG, its unification mechanism is different from PROLOG's syntactic unification. In the following section we introduce SEMLOG's unification mechanism and compare it with its PROLOG counterpart.

4.2 Semantic Unification

The matching of two literals is formally referred to as *semantic unification*. The result of semantic unification is a *list* of substitutions. Each substitution when applied to the two literals make them *semantically equivalent*.

A *substitution* is a finite list of the form $\{(v_1, \tau_1) = e_1, \ldots, (v_n, \tau_n) = e_n\}$, where each element $(v_i, \tau_i) = e_i$ is a *binding* of a logic variable named v_i, of type τ_i, to an expression e_i. For a given substitution, the logic variables in the substitution are distinct. The e_i's in the bindings are restricted to ground expressions and logic variables. For the sake of brevity, in some contexts the type of the logic variable will be omitted from the substitution.

Let $q_4(M, N)$ and $q_4(P, Q.b)$ be two given literals. The signature of the predicate q_4 is $[a : \alpha], \beta$. The goal of semantic unification is to find the substitutions that render the two literals semantically equivalent. For the two literals, $q_4(M, N)$ and $q_4(P, Q.b)$, to be semantically equivalent their predicates must be identical and their respective arguments must match, i.e., they must be semantically equivalent.

Since M and P are both logic variables, the only requirement for the two to be semantically equivalent is that they are bound to the same object. Thus the substitution $\{M = P\}$ makes the two literals semantically equivalent in their first arguments. For the moment, we will ignore the situation when the type of one logic variable is a subtype of the other. We will return to this in a subsequent section.

For the second argument, N and $Q.b$, to be semantically equivalent Q must be bound to a record which has a field component labeled b. In addition the type of the field must be consistent with the type of N. The objects that satisfy these criteria are records of type $[b : \beta]$, where β is the type of the logic variable N. Every object of type $[b : \beta]$, when bound to Q, makes the two expression, N and $Q.b$, semantically equivalent. Thus we have a list of substitutions, one for every object of type $[b : \beta]$. The corresponding binding to N will be the field component labeled b of the record that is bound to Q. Once again, we will ignore for the moment the situation when the type of N is a subtype of the type of $Q.b$.

What is the domain of the logic variable Q? What objects can Q be bound to? Clearly, if we are to consider a universe of all possible values, including all integers, strings, records, etc., we could have an uncountable number of objects of type $[b : \beta]$, and thus an uncountable number of substitutions that make N and $Q.b$ semantically equivalent. But, in SEMLOG, we deal with a universe that is more restrictive, one that contains a fixed number of objects.

In the next section we will describe how we build and maintain this universe of objects. For the moment, let us assume that we have two objects in our universe of type $[b : \beta]$ — $[b := o_3]$ and $[b := o_4]$. The substitutions $\{M = P, Q = [b := o_3], N = o_3\}$ and $\{M = P, Q = [b := o_4], N = o_4\}$ are the result of semantically unifying the two literals $q_4(M, N)$ and $q_4(P, Q.b)$.

The unification mechanism described above is different from PROLOG, where unification is strictly a syntactic process. The result of unifying two terms in PROLOG is a single substitution, not a list of substitutions. The resultant substitution when applied to the two terms make them syntactically identical, not semantically equivalent. For example, syntactic unification can unify the two terms $+(3, X)$ and $+(3, 5)$, but not the terms $+(3, X)$ and $+(2, 6)$ even though we know that if X were bound to 5 the two expressions would be semantically equivalent.

Before we present the algorithm for semantic unification, we discuss the type-object lattice, which plays a crucial role in the semantic unification process.

4.3 The Type-Object Lattice

Unlike PROLOG, where logic variables are universally quantified over all possible terms in the language, logic variables in SEMLOG can only be bound to objects from their domain. The domain of a logic variable is all the objects in the universe of type τ, where τ is the type of the logic variable.

As we had stated earlier, if we are to consider a universe of all possible values we would have an uncountable number of objects of a particular type, say type τ. Instead we restrict our universe to a fixed number of objects.

At the start of an interactive programming session in SEMLOG the universe in empty. The universe is filled with objects that are encountered in the second phase of programming. This is the phase when the user enters knowledge about a particular instance of the problem domain (not to be confused with the domain of a type). In this phase as expressions are encountered they are evaluated and the resultant objects are added to the universe. For example, when the user enters the following fact:

$$\textbf{fact}\ \ p(o_3, [a := o_1])$$

the expressions o_3 and $[a := o_1]$ are evaluated and three new objects are added to the universe — the object o_3, the object $[a := o_1]$ and its subobject o_1.

The universe of objects that is built during the second phase is used during the third phase — the querying phase — to determine the objects that can be bound to a logic variable. Since the type of the logic variable determines the objects that can be bound to it, the objects are stored in the universe on the basis of their types. The data structure that is used to maintain this universe of objects is called the type-object lattice. Before we go into details about the structure of the type-object lattice we first take care of some preliminaries.

4.3.1 Types

Besides the base types — *bool*, *int*, and *string* — the language provides structured types. Structured types are formed by means of type constructors. The type constructors in the language include record types, variant types, and function spaces ($\rightarrow$). In addition to these types there are the two constant types *Top* ($\top$) and *Bottom* ($\bot$) in the language. The type Top represents the type of all values in the language, i.e., every value is of type Top. The type Bottom denotes the inconsistent type, i.e., there are no values of this type.

We present a number of algorithms in this chapter as part of the semantics of the language. In the algorithms we use the following mutually recursive type declarations to denote the abstract syntax of SEMLOG types:

```
datatype TYPE =
   Top                                  |
   Bottom                               |
   Base of string                       |
   RecordType of (LABEL * TYPE) list    |
   VariantType of (LABEL * TYPE) list   |
   Arrow of TYPE * TYPE;
```

Types in SEMLOG are related to one another by the subtype relation. This relation is defined in figure 6.

In addition, for any two types τ and σ, we define the relation "less than" as follows:

- $\tau < \sigma$ if $\tau \leq \sigma$ and $\tau \neq \sigma$

BASIC TYPE

$\iota \leq \iota$

RECORD TYPE

$[a_i : \tau_i, a_j : \tau_j] \leq [a_i : \tau_i'] \iff \tau_i \leq \tau_i' \quad (i \in 1...n, n \geq 0; j \in 1...m, m \geq 0)$

VARIANT RECORD TYPE

$\{a_i : \tau_i\} \leq \{a_i : \tau_i', a_j : \tau_j'\} \iff \tau_i \leq \tau_i' \quad (i \in 1...n, n \geq 0; j \in 1...m, m \geq 0)$

FUNCTION SPACES

$\delta \rightarrow \rho \leq \delta' \rightarrow \rho' \iff \delta' \leq \delta$ and $\rho \leq \rho'$

Figure 6: Subtype Definitions

4.3.2 Expressions

The algorithms that we present here, as part of the semantics of the language, use the the following mutually recursive type declarations to denote the abstract syntax of SEMLOG expressions:

```
datatype EXPR =
      Boolean of bool                              |
      Integer of int                               |
      String of string                             |
      Id of IDENT                                  |
      Record of (LABEL * EXPR) list                |
      Variant of LABEL * EXPR                      |
      LogicVariable of VARIABLE * LABEL * TYPE     |
      Field of EXPR * LABEL                        |
      Case of EXPR * (LABEL * IDENT * EXPR) list   |
      Lambda of IDENT * TYPE * EXPR                |
      App of EXPR * EXPR;
```

Given an expression e in the language, the function TYPE_OF_EXPR determine the type of the expression from a type environment for identifiers, i.e., a mapping from identifiers to types. The following basic operations for the type environment $\mathcal{T}$ are assumed:

- *EMPTY_TYPE_ENV* — yields an empty type environment.
- *VALUE_TYPE_ENV*$(i, \mathcal{T})$ — yields the type bound to identifier i in the type environment $\mathcal{T}$.
- *UPDATE_TYPE_ENV*$(i, \tau, \mathcal{T})$ — yields the type environment obtained by adding to the type environment $\mathcal{T}$ a binding of identifier i to the type τ.

4.3.3 The Type Lattice

In SEMLOG, the set of all types when ordered by the "less than" relation forms a partial order. The partial order can be represented graphically as a lattice. We usually represent some finite portion of the subtype relation by a *Hasse diagram*. A Hasse diagram is a graph structure whose nodes represent the type elements and whose edges are directed downward, from node τ to node η if η is *covered* by τ. A type η is *covered* by type τ, or τ *covers* η, if $\eta < \tau$ and there is no type γ in the Hasse diagram such that $\eta < \gamma < \tau$. The symbols $\ll$ and $\gg$ are used to represent the relations covered and cover, respectively. Figure 9 is a Hasse diagram of a sample portion of the type lattice. In Figure 9, $[a : int; b : string; c : bool] \ll [c : bool]$, because there is no type element γ in the lattice such that $[a : int; b : string; c : bool] < \gamma < [c : bool]$.

Since the type $\bot$ is less than every other type in the lattice, the node that represents the type $\bot$ occupies the lowest position in the lattice. Dually, since every other type in the lattice is less than the type $\top$, the

BASIC TYPE

$\iota \otimes \iota = \iota$

RECORD TYPE

$[a_i : \tau_i, b_j : \sigma_j] \otimes [a_i : \tau_i', c_k : \eta_k] = [a_i : \tau_i \otimes \tau_i', b_j : \sigma_j, c_k : \eta_k] \quad (\forall j,k\ b_j \neq c_k)$

VARIANT RECORD TYPE

$\{a_i : \tau_i, b_j : \sigma_j\} \otimes \{a_i : \tau_i', c_k : \eta_k\} = \{a_i : \tau_i \otimes \tau_i'\} \quad (\forall j,k\ b_j \neq c_k)$

FUNCTION SPACES

$(\delta \rightarrow \rho) \otimes (\delta' \rightarrow \rho') = (\delta \oplus \delta') \rightarrow (\rho \otimes \rho')$

OTHERWISE

$\tau \otimes \tau' = \bot$

Figure 7: MEET Operation

top of the lattice is the node denoting the type $\top$. Because the subtype relation is transitive, we omit the edge from τ to σ if there is another path from τ to σ in the lattice. For example, in Figure 9 we omit the edge from $\top$ to $[a : int; b : string]$ even though $[a : int; b : string] < \top$, since there is another path from $\top$ to $[a : int; b : string]$ through $[a : int]$.

Given two types τ and σ in the lattice, the *meet* of τ and σ, denoted by $\tau \otimes \sigma$, is the highest node, type η, for which there is a path downward to η from both τ and σ. For example, in Figure 9, the meet of $[a : int]$ and $[b : string]$ is the type $[a : int; b : string]$.

Dually, we can define the *join* of two types τ and σ. The join of τ and σ, denoted by $\tau \oplus \sigma$, is the lowest node in the type lattice, type γ, for which there is a path downward from γ to both τ and σ. For example, in the Figure 8, the join of $[a : int; b : string]$ and $[b : string; c : bool]$ is the type $[b : string]$.

The meet and join operations for the various type constructors are shown in Figure 7 and 8, respectively.

BASIC TYPE

$\iota \oplus \iota = \iota$

RECORD TYPE

$[a_i : \tau_i, b_j : \sigma_j] \oplus [a_i : \tau_i', c_k : \eta_k] = [a_i : \tau_i \oplus \tau_i'] \quad (\forall j,k\ b_j \neq c_k)$

VARIANT RECORD TYPE

$\{a_i : \tau_i, b_j : \sigma_j\} \oplus \{a_i : \tau_i', c_k : \eta_k\} = \{a_i : \tau_i \oplus \tau_i', b_j : \sigma_j, c_k : \eta_k\} \quad (\forall j,k\ b_j \neq c_k)$

FUNCTION SPACES

$(\delta \rightarrow \rho) \oplus (\delta' \rightarrow \rho') = (\delta \otimes \delta') \rightarrow (\rho \oplus \rho')$

OTHERWISE

$\tau \oplus \tau' = \top$

Figure 8: JOIN Operation

In SEMLOG, at the start of an interactive programming session, the lattice contains only one type element, the type $\top$; no objects are stored in the lattice. During the second phase as objects are encountered they are added to the the lattice. The objects are stored in the lattice on the basis of their types. The type of the object determines the node in the lattice where it is stored. If the type of the object is not present in the lattice, the addition of the type must precede the addition of the object.

The ML data types for the type-object lattice are given below. The type-object lattice for SEMLOG is represented by the ML object `TypeLattice`. It initially contains only the single type `Top`.

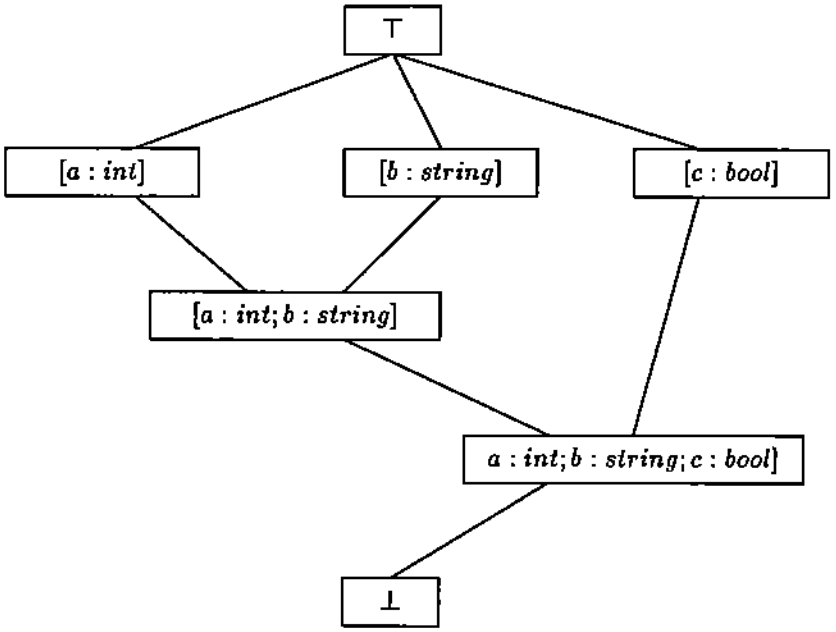

Figure 9: Type lattice

```
  (************************************************************************
     LATTICE NODE = Type * Mark * Exprs * Subnodes
   ************************************************************************)

datatype LATTICE =
  Lattice of TYPE * (bool ref) * (EXPR Set ref) * (LATTICE ref Set ref);

type Subnodes = LATTICE ref Set;

  (************************************************************************
     Initialize the LATTICE by creating the Top Node.
   ************************************************************************)

val TypeLattice =
  [ref (Lattice (Top, ref false, ref EmptySet, ref EmptySet))] ;
```

Algorithms for determining the existence of a type, adding a non-existent type, and adding an object are presented in the next few sections. In the algorithms, we use τ° to denote the set of objects that are contained in the node representing type τ. For any type η, we use $\eta^{\ll}$ to denote the set of type elements in the lattice that cover type η, and $\eta^{\gg}$ to denote the set of type elements in the lattice that are covered by η.

Let $\mathcal{L}$ represent the set of nodes in the lattice. For any type η, we formally define $\eta^{\ll}$ and $\eta^{\gg}$ as follows:

$$\eta^{\ll} = \{\gamma \in \mathcal{L} \mid \eta \ll \gamma\}$$
$$\eta^{\gg} = \{\gamma \in \mathcal{L} \mid \eta \gg \gamma\}$$

The set of type elements in the lattice that cover a type η, $\eta^{\ll}$, is also called the *coverset* of the type.

It is not necessary for the type η to occur in the lattice. If the type η occurs in the lattice, then $\eta^{\ll}$ represents the set of nodes that have directed edges *to* the node representing type η. The set of nodes that have directed edges *from* the node representing type η is denoted by $\eta^{\gg}$.

4.3.4 Determining the Existence of a Type in a Lattice

We determine the existence of a type by searching the lattice structure. The search procedure starts at the top node of the lattice, the node representing the type $\top$, and follows a path to the type element, if the type element exists. The search is terminated upon finding the type element, or if a dead end is reached.

The ML algorithm for determining the existence of a type in a lattice is the function EXIST given below. It is invoked at the top level as EXIST(η, $\{\top\}$).

Let η denote the type whose existence is to be determined. The search procedure starts at the top of the lattice and proceeds as follows. Let σ denote the type of the most recent node visited. If $\eta = \sigma$ then we have found the type and the search is terminated. Otherwise, the next node to be visited is determined as follows. From the set $\sigma^{\gg}$, the set of type elements covered by σ, we determine a type γ such that $\gamma > \eta$, i.e., $\gamma \in \sigma^{\gg}$ and $\gamma > \eta$. The type element γ is the next node to be visited. If no such γ exists, then we have reached a dead end, and we can conclude that the type does not exist in the lattice.

The algorithm terminates because the lattice is finite and a different node is visited at every step in the recursion.

```
fun Exist (eta: TYPE, LatticeNodes: Subnodes): bool =
  if IsEmptySet LatticeNodes then false
  else
  let
    val SetElement = AnyElement LatticeNodes;
    val Lattice (sigma,Mark,Exprs,SubTypeSet) = !SetElement;
    val phi = MEET(sigma,eta)
  in
    if sigma = eta then true
    else
        if phi = eta (* sigma > eta *) then
                  Exist(eta, !SubTypeSet)
        else
           Exist(eta, Delete(SetElement,LatticeNodes))
  end;
```

4.3.5 Adding a Type to a Lattice

During the course of SEMLOG program new objects will be introduced and these must be placed in the type-object lattice. The first step is to add a type to the lattice. Consider adding the type $[b : string; c : bool]$ to the lattice in Figure 9. The type $[a : int; b : string; c : bool]$ is no longer covered by $[c : bool]$, and therefore the edge in the lattice that represents this cover relation will have to be destroyed. The addition of the type introduces some new edges. These edges are illustrated by dotted arrows in Figure 10.

The lattice has to be reconfigured to accommodate the new type. Edges representing cover relations that are no longer true will have to be destroyed and new edges will have to be created to represent the cover relations that are introduced in the lattice.

Let η denote the type that is to be added to the lattice. The coverset of η, denoted by $\eta^{\ll}$, is the set of type elements in the lattice that cover η. We create directed edges from the elements of $\eta^{\ll}$ to η to represent these new cover relations. The algorithm to determine the coverset η is presented in the next section.

For every $\sigma \in \eta^{\ll}$, we define $\mathcal{C}_{\eta,\sigma}$ as follows:

$$\mathcal{C}_{\eta,\sigma} = \{\tau \in \sigma^{\gg} \mid \tau < \eta\}$$

The set $\mathcal{C}_{\eta,\sigma}$ represents the lattice elements that are covered by σ, but less than η. In other words, the cover relation between σ and the elements of $\mathcal{C}_{\eta,\sigma}$ will no longer be true following the addition of the type η. The edges that represent these cover relations will have to be destroyed.

The elements of $\mathcal{C}_{\eta,\sigma}$ are covered by η instead. Edges reflecting this fact will have to be created. These newly created edges replace the edges that are destroyed.

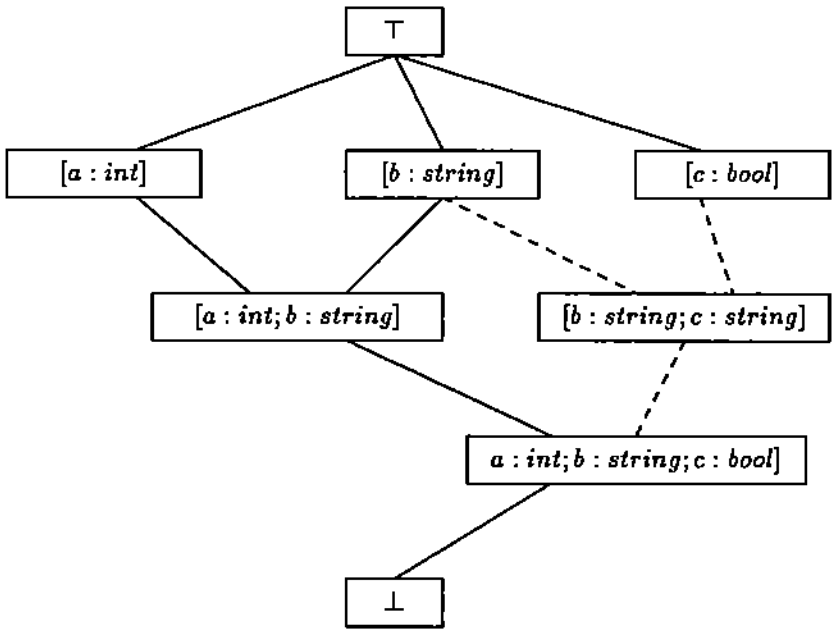

Figure 10: Adding a type to the lattice

```
fun Add_Type(eta:TYPE, TypeLattice):unit =
   if not (Exist(eta,Set TypeLattice)) then
      let
        val NewNode = ref(Lattice(eta,(ref false),ref EmptySet,ref EmptySet));
        val cover_set_of_eta = Cover_set(eta,Set TypeLattice)
      in ReConfigureLattice(NewNode, cover_set_of_eta)
      end
   else ()  ;
```

The steps involved in adding a type η to a lattice are summarized below:

1. Create a node to be included in the lattice structure. The node will represent the type η.
2. Determine the coverset of η. This set is denoted by $\eta^{\ll}$. For every $\sigma \in \eta^{\ll}$ create a directed edge from σ to η.
3. For every $\sigma \in \eta^{\ll}$, determine the set of $\mathcal{C}_{\eta,\sigma}$, and perform the following operations:
   (a) Destroy the edges between σ and the elements of $\mathcal{C}_{\eta,\sigma}$.
   (b) Create directed edges from η to the elements of $\mathcal{C}_{\eta,\sigma}$.

The procedure ReConfigureLattice, shown below performs the reconfiguration of the lattice. The function LESS computes the set $\mathcal{C}_{\eta,\sigma}$.

```
fun ReConfigureLattice (NewType: LATTICE ref, Coverset:Subnodes) =
  if IsEmptySet Coverset then ()
  else
    let
      val Lattice (eta, NewMark, NewExprs, NewSubTypeSet) = !NewType;
      val SetElement = AnyElement Coverset;
```

```
      val Lattice (sigma, Mark, Exprs, SubTypeSet) = !SetElement;
      val C_eta_sigma = Less(eta, !SubTypeSet)
    in
      SubTypeSet := (Add(NewType, Difference(!SubTypeSet, C_eta_sigma)));
      NewSubTypeSet := (Union(C_eta_sigma, !NewSubTypeSet));
      ReConfigureLattice (NewType, Delete(SetElement, Coverset))
    end;

fun Less(eta: TYPE, LatticeNodes: Subnodes): Subnodes =
  if IsEmptySet LatticeNodes then EmptySet
  else
    let
      val SetElement = AnyElement LatticeNodes;
      val Lattice (tau, Mark, Exprs, SubTypeSet) = !SetElement;
      val phi = MEET (tau, eta);
      val LESS = Less(eta, Delete(SetElement,LatticeNodes))
    in
      if phi = tau then Add(SetElement,LESS)
      else LESS
    end;
```

4.3.6 Determining the Coverset of a Type

We present in this section the algorithm to determine the coverset of a type. The algorithm assumes that the type whose coverset is being determined does not occur in the lattice. Throughout the discussion, the type whose coverset is being determined will be denoted by η, and its coverset will be denoted by $\eta^{\ll}$. Let:

$$H_{\eta,\alpha} = \{\beta \in \alpha^{\gg} \mid \beta > \eta\}$$

For a given lattice type element α, the set $H_{\eta,\alpha}$ denotes the set of type elements covered by α that are greater than η. If the set $H_{\eta,\alpha}$ is empty then clearly α is a cover of η. If not, every element of $H_{\eta,\alpha}$ is a candidate for being a cover of η.

The set $H_{\eta,\alpha}$ defined above is used in the following recurrence equations to determine the coverset of a type:

$$\mathcal{S}_{\eta,n} = \begin{cases} \{\mathsf{T}\} & \text{for } n = 0 \\ \bigcup_{\alpha \in \mathcal{S}_{\eta,n-1}} H_{\eta,\alpha} & \text{for } n > 0 \end{cases}$$

$$\mathcal{A}_{\eta,n} = \{\alpha \in \mathcal{S}_{\eta,n-1} \mid H_{\eta,\alpha} = \emptyset\} \quad \text{for } n > 0$$

$$\eta_n^{\ll} = \begin{cases} \emptyset & \text{for } n = 0 \\ \eta_{n-1}^{\ll} \cup \mathcal{A}_{\eta,n} & \text{for } n > 0 \end{cases}$$

At any step k in the recursion, the set $\mathcal{S}_{\eta,k}$ denotes the set of elements that are candidates to be covers of η. Each element of the set $\mathcal{S}_{\eta,k}$ is a cover candidate, a possible cover of η. Initially, the set $\mathcal{S}_{\eta,0}$ contains the single lattice element T. Since the element T is the greatest element in the type lattice, it is the first element to be considered as a possible cover.

We compute $\mathcal{S}_{\eta,n}$ from $\mathcal{S}_{\eta,n-1}$ as follows. For every $\alpha \in \mathcal{S}_{\eta,n-1}$, the cover candidates generated from the previous recursion step, we determine whether α is a cover of η. If α is not a cover of η then there exists at least one lattice element covered by α that is greater than η, and hence a cover candidate for the next recursion step. Of the lattice elements that are covered by α ($\alpha^{\gg}$), the ones that are greater than η are cover candidates for the next recursion step, the elements of the set $\mathcal{S}_{\eta,n}$.

```
fun Cover_set (sigma: TYPE, S_sigma_n: Subnodes): Subnodes =
  let
    val (M_sigma_n_plus_1, S_sigma_n_plus_1) =
      COVER_CAN (sigma, S_sigma_n)
  in
    if IsEmptySet S_sigma_n_plus_1 then M_sigma_n_plus_1
    else Union(M_sigma_n_plus_1, Cover_set(sigma, S_sigma_n_plus_1))
  end;
```

The set $\mathcal{A}_{\eta,n}$ denotes the covers identified in step n. To identify these covers we determine for every $\alpha \in \mathcal{S}_{\eta,n-1}$, the cover candidates generated from the previous step, whether α is a cover of η. If α is a cover of η it is added to the set $\mathcal{A}_{\eta,n}$.

At any step n, the set $\eta_n^{\preccurlyeq}$ denotes all the covers of η identified thus far. This includes the covers identified in step n. Initially, the set $\eta_0^{\preccurlyeq}$ is the empty set. As new covers are found they are added to this set.

At any step k, the cover candidates $\mathcal{S}_{\eta,k}$ are the nodes in the lattice that are k edges away from the top of the lattice. The terminating condition for the recursion is when there are no more cover candidates, i.e., $\mathcal{S}_{\eta,m}$ is empty for some $m > 0$. The recursion terminates because the lattice is finite and different nodes are visited at every step of the recursion.

The algorithm for determining the coverset of a type is shown below. The function `COVER_CAN` computes $\mathcal{S}_{\eta,n}$ and $\mathcal{A}_{\eta,n}$ from $\mathcal{S}_{\eta,n-1}$. The function `Greater` computes the set $H_{\eta,\alpha}$ and is omitted since it is similar to `Less`

```
fun COVER_CAN (sigma: TYPE, S_sigma_n: Subnodes): Subnodes * Subnodes =
  if IsEmptySet S_sigma_n then (EmptySet, EmptySet)
  else
    let
      val SetElement = AnyElement S_sigma_n;
      val Lattice (tau, Mark, Exprs, SubTypeSet) = !SetElement;
      val GREATER = Greater(sigma, !SubTypeSet);
      val (Minimal, S_sigma_n_plus_1) =
        COVER_CAN(sigma, Delete(SetElement,S_sigma_n))
    in
      if IsEmptySet GREATER then (Add(SetElement, Minimal), S_sigma_n_plus_1)
      else (Minimal, Union(GREATER, S_sigma_n_plus_1))
    end;
```

4.3.7 Adding an Object to a Lattice

The algorithm for adding an object to a lattice is shown below. Notice that the procedure `Add_Type` is invoked from `Add_Object` to add the type of the object. The procedure `Add_Type` adds the type only if it is not already present in the lattice. During the execution of a SEMLOG program it is necessary to be able to find all objects of a given type. Thus at run time a type lattice is maintained. If a new object is introduced, then its type must point to it.

```
fun Add_Object(obj: EXPR, eta: TYPE): unit =
  if Add_Type(eta, TypeLattice)= () then
  let
    val Lattice (tau, Mark, Exprs, SubTypeSet) = !(FindTypeNode(eta,hd TypeLattice))
  in
    if Member(obj, !Exprs) then ()
    else     (*Insert object*)
Exprs := Add(obj, !Exprs)
  end
  else raise woops ;
```

4.3.8 Finding All Objects of a Type

The lattice is used during semantic unification to determine the objects that can be bound to a logic variable. Since the type of the variable determines the objects that can be bound to it, finding all objects in the lattice of a type is crucial to the semantic unification algorithm.

Given a type η, what does it mean to find all objects of the type? Clearly, any object in the lattice element η is a valid object. But, what about an object that is stored in the lattice element τ, where τ is a subtype of η? In SEMLOG, subtyping allows the substitution of an object of the subtype, wherever an object of the type can occur. In other words, an object of type τ is also an object of type η, if τ is a subtype of η. Thus any object whose type is a subtype of η is also a valid object.

Finding all objects in the lattice of type η involves visiting not only the element η but also the elements in the lattice which are subtypes of η. We do this by starting at the lattice element η and visiting all the lattice elements that are reachable from η. We perform a depth first search of the lattice structure starting at the node η. We mark the nodes as we visit them so as to avoid revisiting nodes.

The algorithm for finding all the objects of a type is given below. The function `Find_Objects` invokes the function `Add_Type` to add the type η to the lattice. The type must occur in the lattice for the function to work correctly.

```
fun FindObjects(eta: TYPE): EXPR Set =
  let
    val TYPENODE = FindTypeNode(eta, hd TypeLattice);
    val ALLEXPRS = TypeObjects(TYPENODE)          (*Find the Exprs*)
  in
    ( UNMARK(TYPENODE);                           (*Unmark the visited nodes*)
      ALLEXPRS  )                                 (*Return the Exprs*)
  end;

fun TypeObjects(Node: LATTICE ref): EXPR Set =
  let
    val Lattice(tau, Mark, Exprs, SubTypeSet) = !Node
  in
    if (!Mark) then EmptySet
    else
      let
        val SUBTYPEOBJECTS = SubTypeObjects(!SubTypeSet)
      in
        ( Mark := true;
          Union(!Exprs, SUBTYPEOBJECTS) )
      end
  end

and

fun SubTypeObjects (LatticeNodes: Subnodes): EXPR Set =
  if IsEmptySet LatticeNodes then EmptySet
  else
    let
      val SetElement = AnyElement LatticeNodes;
      val TYPEOBJECTS = TypeObjects(SetElement);
      val RESTOBJECTS = SubTypeObjects(Delete(SetElement,LatticeNodes))
    in
      Union(TYPEOBJECTS, RESTOBJECTS)
    end;
```

4.4 Semantic Unification Algorithm

In Section 4.2 we introduced SEMLOG's unification mechanism. In this section we present the algorithm for semantic unification.

Given two expressions, the algorithm determines if they are semantically equivalent. It returns a list of substitutions if unification is successful, otherwise it fails. In case unification is successful, the list of substitutions returned is such that each substitution in the list when applied to the two expressions make them semantically equivalent.

The semantic unification algorithm is given below. The algorithm assumes the existence of the type-object lattice — the universe of objects. In the algorithm p and q are the two expressions to be unified. $\mathcal{S}$ denotes the substitution prior to unification, i.e., the current substitution. The expressions p and q are assumed to be in normal form. In addition, the algorithm assumes the following basic operations for the substitution $\mathcal{S}$

- *EMPTY_SUBST* — yields an empty substitution.
- *VALUE_SUBST*$((v,\tau),\mathcal{S})$ — yields the expression bound to the logic variable (v,τ) in the substitution $\mathcal{S}$. If the variable is bound to another variable, (v',τ'), we recursively invoke *VALUE_SUBST* with (v',τ'). If the variable is not bound to anything in the substitution it returns itself.
- *UPDATE_SUBST*$((v,\tau), e, \mathcal{S})$ — yields the substitution obtained by adding to the substitution $\mathcal{S}$ the binding $(v,\tau) = e$.

The two expressions, p and q, that are to be unified may contain *free* variables. Free variables are logic variables that are unbound in the current substitution. Variables that have bindings in $\mathcal{S}$ are known to be *bound*.

Based on the form of the expressions p and q, the semantic unification algorithm is divided into the following four cases:

1. p and q are both logic variables.
2. p is a logic variable, but q is any expression other than a logic variable.
3. q is a logic variable, but p is any expression other than a logic variable.
4. Both p and q are expressions, but neither is a logic variable.

Next we describe the individual cases in detail. The algorithm SEMANTIC_UNIFY in ML is given in full after the description.

Case 1: Let p be a logic variable (p_v,τ), where p_v is the name of the variable and τ its type, and q a logic variable (q_v,σ), where q_v is the name and σ the type. If both p and q are of the same type, i.e., $\tau = \sigma$, then any object from the domain of the logic variable p (which is also the domain of q) can be bound to both variables making them semantically equivalent. Since the variables, p and q, are both of the same type their domains are identical. The domain of the logic variable p is all the objects in the type-object lattice of type τ.

The substitution that would make p and q semantically equivalent is one that contains the binding $(p_v,\tau) = (q_v,\sigma)$. Thus the new substitution, $\mathcal{S}_n$, is obtained by adding this new binding to the current substitution, $\mathcal{S}$.

If $\tau \neq \sigma$, then the only objects that can be bound to both variables are objects from the intersecting domain. The intersecting domain is all the objects in the type-object lattice of type δ, where $\delta = MEET(\tau,\sigma)$. If $\delta = \perp$, then unification fails, because there are no objects of type $\perp$ in the lattice.

If $\tau \neq \sigma$ and if $\delta = MEET(\tau,\sigma)$, then the new substitution $\mathcal{S}_n$ is obtained by adding to the current substitution $\mathcal{S}$, the bindings $(p_v,\tau) = (q_v,\delta)$ and $(q_v,\sigma) = (q_v,\delta)$. The reason for including the binding $(q_v,\sigma) = (q_v,\delta)$ is because the type of the logic variable q has been constrained to a subtype of its original type, as a result of unifying p and q.

Case 2: p is a logic variable and q is any expression which is not a logic variable. Let p be the logic variable (p_v, τ). The expression q may contain free variables. Let $\mathcal{F}$ represent the set of free variables in q. Given an expression e, the function *EXPR_FREE_VARS* determines the set of free variables in the expression. There are two potential cases to be considered.

In the first case, $\mathcal{F} = \emptyset$, i.e., q has no free variables. For p and q to be unifiable: the type of q must be a subtype of the logic variable p. The new substitution, $\mathcal{S}_n$, is obtained by adding the binding $(p_v, \tau) = q$ to the current substitution $\mathcal{S}$. The result of semantic unification in this case is a single substitution.

Next, we consider the case when $\mathcal{F} \neq \emptyset$, i.e., q contains free variables. Since a free variable is a variable that is not bound to anything in $\mathcal{S}$, it can be instantiated to any object from its domain. The domain of a free variable is all the objects in the type-object lattice of type τ, where τ is the type of the free variable.

For p and q to be unifiable: (1) the free variables in q must be bound to objects from their respective domains and (2) the type of object o must be a subtype of the type of the logic variable p, where o is the result of evaluating the expression q after all its free variables have been instantiated. The new substitution is obtained by adding the binding $(p_v, \tau) = o$ to $\mathcal{S}@\mathcal{S}_f$, where $\mathcal{S}$ is the current substitution, $\mathcal{S}_f$ the binding of free variables to objects, and "@" the operator that appends two substitution lists.

There is a list of these new substitutions that make p and q unifiable — potentially as many as all possible combinations of free variable-object bindings. For example, let us assume that the expression q contains two free variables — U and V. Let the domain of U contain the objects o_1 and o_2, and the domain of V the objects o_3 and o_4. The possible combinations of free variable-object bindings are: $\{U = o_1, V = o_3\}, \{U = o_1, V = o_4\}, \{U = o_2, V = o_3\}, \{U = o_2, V = o_4\}$.

In order to facilitate the description of this part of the algorithm we first introduce some notation. Let the free variable set:

$$\mathcal{F} = \{(v_1, \tau_1), \ldots, (v_i, \tau_i), \ldots, (v_n, \tau_n)\}$$

where each (v_i, τ_i) is a free variable, v_i the name of the free variable and τ_i its type. Let the domain of the free variables be represented by:

$$\mathcal{D}(\tau_i) \quad \forall i \in 1 \ldots n$$

Let:

$$\mathcal{S}_j = \{(v_1, \tau_1) = o_1, \ldots, (v_n, \tau_n) = o_n\}$$
$$o_1 \in \mathcal{D}(\tau_i), \ldots, o_n \in \mathcal{D}(\tau_n)$$

represent a substitution obtained by binding the free variables to objects from their respective domains. We can obtain a list of such substitutions — one for each combination of free variable-object bindings. The list of such substitutions will be denoted by $\mathcal{S}_l$. The function *FREE_VARS_TO_SUBSTS* to determine $\mathcal{S}_l$ in *SEMANTIC_UNIFY* is assumed. From $\mathcal{S}_l$ we determine the new substitution list as follows.

Let $\mathcal{S}_j$ be an element of $\mathcal{S}_l$, and o_j the object that results from evaluating the expression q using the substitution $\mathcal{S}_j$.

Let F be a function that, given $\mathcal{S}_j$, returns the pair $(o_j, \mathcal{S}_j)$. Let G be a function that returns true if the type of o_j is a subtype of the type of the logic variable p, otherwise it returns false. Let H be a function that, given the pair $(o_j, \mathcal{S}_j)$, returns the substitution obtained by adding the binding $(p_v, \tau) = o_j$ to $\mathcal{S}@\mathcal{S}_j$, where $\mathcal{S}$ is the current substitution. The list of new substitutions is obtained by:

$$(\mathbf{map} \quad H) \circ (\mathbf{filter} \quad G) \circ (\mathbf{map} \quad F) \quad \mathcal{S}_l$$

where $\circ$ denotes the composition of functions, **map** and **filter** represent the higher-order functions with their usual meaning.

The function *NEW_SUBST_LIST* determines the list of new substitutions that make p and q unifiable. This function is invoked from within *SEMANTIC_UNIFY*.

In function *NEW_SUBST_LIST* the function *EVAL_EXPR_WITH_VARS*, evaluates the the expression e using the substitution $\mathcal{S}_j$.

Case 3: this case is similar to Case 2.

Case 4: neither p nor q is a logic variable. Both expressions, p and q, may contain free variables. Let $\mathcal{F}_p$ represent the free variables in p, and $\mathcal{F}_q$ the free variables in q. Let $\mathcal{F}$ represent $\mathcal{F}_p \bigcup \mathcal{F}_q$.

Let us first consider the case when $\mathcal{F} = \emptyset$, i.e., both p and q have no free variables. For p and q to be unifiable: the two expressions p and q must be identical, i.e., they must both denote the same object. The new substitution, $\mathcal{S}_n$, is identical to the current substitution $\mathcal{S}$. Since there are no free variables in p and q, no new bindings are formed.

Next we consider the case when $\mathcal{F} \neq \emptyset$, i.e., either p or q, or both p and q contain free variables. Since a free variable is a variable that is not bound to anything in $\mathcal{S}$, it can be instantiated to any object from its domain. For p and q to be unifiable: (1) the free variables in p and q must be bound to objects from their respective domains (2) the objects o_p and o_q must be identical, where o_p and o_q are the objects that result from evaluating p and q respectively, after all their free variables have been instantiated. The new substitution is $\mathcal{S}@\mathcal{S}_f$, where $\mathcal{S}$ is the current substitution, $\mathcal{S}_f$ the binding of free variables to objects, and "@" the operator that appends two substitution lists.

There is a list of these new substitutions that make p and q unifiable — potentially as many as all possible combinations of free variable-object bindings. To determine this list of new substitutions, we need $\mathcal{S}_l$.

The list $\mathcal{S}_l$ is the list of all possible combinations of free variable-object bindings. As was stated in Case (2), the algorithm to determine the list $\mathcal{S}_l$ is assumed. From $\mathcal{S}_l$ we determine the new substitution list as follows.

Let $\mathcal{S}_j$ be an element of $\mathcal{S}_l$, and o_p^j and o_q^j the objects that result from evaluating the expressions p and q, respectively, using the substitution $\mathcal{S}_j$. Let F be a function that, given $\mathcal{S}_j$, returns the triple $(o_p^j, o_q^j, \mathcal{S}_j)$. Let G be a function that returns true if the objects o_p^j and o_p^j are identical, otherwise it returns false. Let H be a function that, given the triple $(o_p^j, o_q^j, \mathcal{S}_j)$, returns the substitution $\mathcal{S}@\mathcal{S}_j$, where $\mathcal{S}$ is the current substitution. The list of new substitutions is obtained by:

$$(\textbf{map}\ \ H) \circ (\textbf{filter}\ \ G) \circ (\textbf{map}\ \ F)\ \ \mathcal{S}_l$$

where $\circ$ denotes the composition of functions, **map** and **filter** represent the higher-order functions with their usual meaning.

The function *NEW_SUBST_LIST_VARIANT* determines the list of new substitutions that make p and q unifiable. This function is invoked from within *SEMANTIC_UNIFY*.

```
fun SemanticUnify(X: EXPR, Y: EXPR, S: SUBST): (bool * SUBST list) =
  let
    val U = InstantiateVar (S) (X);
    val V = InstantiateVar (S) (Y)
  in
    case (U,V) of
      (LogicVariable(VARU,LVLU,TYPU), LogicVariable(VARV,LVLV,TYPV))  =>
         if TYPU = TYPV then
            let
              val NewSubst =
                UpdateSubst((VARU,LVLU,TYPU), LogicVariable(VARV,LVLV,TYPV), S)
            in
              (true, NewSubst::□)
            end
         else
            let
              val tau = MEET(TYPU, TYPV);
              val lv = LogicVariable (VARV, LVLU, tau);
            in
              if tau <> Bottom then
                 let
```

```
            val NewSubst1 = UpdateSubst((VARU,LVLU,TYPU), lv, S);
            val NewSubst = UpdateSubst((VARV,LVLV,TYPV), lv, NewSubst1)
          in
            (true, NewSubst::[])
          end
      else  (false, [])
    end                                |
( LogicVariable(VARU,LVLU,TYPU), _          )          =>
  let
    val FV = FreeVarInExpr(V)
  in
    if IsEmptySet FV then
      if TYPU = JOIN(TYPU,(TypeOfExpr EmptyTypeEnv V)) then
        let
          val NewSubst = UpdateSubst((VARU,LVLU,TYPU),V,S)
        in
          (true, NewSubst::[])
        end
      else  (false, [])
    else
      if Member((VARU,LVLU,TYPU),FV) then
        (false, [])
      else                                  |
        let
          val SubstList = VarsToSubsts(FV);
          val NewSubstList = FormNewSubstList((VARU,LVLU,TYPU),V,S, SubstList)
        in
          if NewSubstList = [] then
            (false, [])
          else
            (true, NewSubstList)
        end
    end                            |

( _             , LogicVariable(VARV,LVLV,TYPV))  =>
    SemanticUnify(V,U,S)  |

( _                 , _           )         =>
      let
        val FVU = FreeVarInExpr(U);
        val FVV = FreeVarInExpr(V);
        val FV = Union(FVU, FVV)
      in
        if IsEmptySet FV then
          if U = V then
              (true, S::[])
          else
              (false, [])
        else
          let
            val SubstList = VarsToSubsts(FV);
            val NewSubstList = FormNewSubstListVariant(U,V,S,SubstList)
```

```
                in
                   if null NewSubstList then
                            (false, [])
                   else
                            (true, NewSubstList)
                 end
            end

  end;

fun FormNewSubstList((v,l,t): (VARIABLE * LEVEL * TYPE),
                     e:EXPR, S:SUBST, SL:SUBST list): SUBST list =
    let
          fun F(S':SUBST) : EXPR*SUBST =
              (EvalWithLogicVars EmptyEnv S' e, S')
      and
             G((obj,S'): EXPR*SUBST) : bool =
                 if t = JOIN(t,TypeOfExpr EmptyTypeEnv obj)
                 then true
                 else false
      and
             H((obj,S'): EXPR*SUBST) : SUBST =
                 UpdateSubst((v,l,t),obj,AppendSubst (S,S'))
    in
             (map H) o (filter G) o (map F) SL
    end;
```

4.5 An Example

In the following example we trace through the interpretation of a query to illustrate SEMLOG's resolution and unification process. The universe of objects is shown in Figure 12. The rules and facts of the example are shown in Figure 11

The query is $q_4(M, N)$. The entire search space of the query is shown in Figure 12. In the search space any path from the original goal to a box with the caption "success" represents a solution. The query yields three substitutions for the variables M and N as solutions. They are:

$$\{M = [a := o_1], N = o_3\}$$
$$\{M = [a := o_2], N = o_3\}$$
$$\{M = [a := o_2], N = o_4\}$$

The way SEMLOG finds these solutions is as follows. SEMLOG searches through the list of clauses and finds that the goal $q_4(M, N)$ unifies the head of Clause(5), $q_4(P, Q.b)$. The result of unifying the two literals, $q_4(M, N)$ and $q_4(P, Q.b)$, is two substitutions. SEMLOG determines the two substitutions as follows.

For the literals, $q_4(M, N)$ and $q_4(P, Q.b)$, to match their predicates must be identical and their respective arguments must match, i.e., they must be semantically equivalent. The substitution $\{M = P\}$ makes the first arguments, M and P, semantically equivalent. Since Q is not bound to anything in the current substitution, $\{M = P\}$, we need to find objects in the domain of Q that when bound to Q will render the two expressions $Q.b$ and N semantically equivalent. There are two objects in the domain of Q. These are the two objects of type $[b : \beta]$. Each object when bound to Q makes the two expressions $Q.b$ and N semantically equivalent. The corresponding binding to N will be the field component labeled b of the object that is bound to Q. The result of unifying $q_4(M, N)$ and $q_4(P, Q.b)$ is the following two substitutions:

$$\{M = P, Q = [b := o_3], N = o_3\}$$
$$\{M = P, Q = [b := o_4], N = o_4\}$$

We can choose either of the two substitutions for the next step. If we reach a dead end by choosing one we can pursue the other. As a result of unification we have a fan-out (branching) in the search space of Figure 12. The degree of the fan-out is the number of substitutions returned. Let us temporarily interrupt our discussion to point out that in PROLOG the result of a successful unification is a single substitution.

Let us choose the substitution:

$$\{M = P, Q = [b := o_4], N = o_4\}$$

for the next step. The goal $q_4(M, N)$ is replaced by the body of Clause(5) and appropriate substitutions are made, yielding the following resolvent:

$$q_1([b := o_4], [d := o_5]), q_2(P.a, [b := o_4])$$

Next we try matching the goal $q_1([b := o_4], [d := o_5])$ in the resolvent. The goal does not unify with the head of any of the clauses in our list of clauses. It does not match with Clause(1) because $[b := o_3]$ and $[b := o_4]$ are two different objects and hence are not semantically equivalent. We have reached a dead end. In the search space dotted boxes with the caption "fail" denote failures that result because one or more of the arguments of the literals did not match, even though the predicates matched.

We backtrack and pursue the other substitution:

$$\{M = P, Q = [b := o_3], N = o_4\}$$

The goal $q_4(M, N)$ is replaced by the body of Clause(5) and appropriate substitutions are made yielding:

$$q_1([b := o_3], [d := o_5]), q_2(P.a, [b := o_3])$$

The goal $q_1([b := o_3], [d := o_5])$ successfully unifies with the head of Clause(1). No new bindings are introduced. As a result the substitution remains the same. Notice that the degree of fan-out in the search space at this point is one. Since the body of Clause(1) is empty, our new resolvent is:

$$q_2(P.a, [b := o_3])$$

We try matching the goal $q_2(P.a, [b := o_3])$, with the head of Clause (2), $q_2(o_1, [b := o_3])$. Since P is not bound to anything in the current substitution, we need to find objects in the domain of P that when bound to P will render the two expression $P.a$ and o_1 semantically equivalent. There are two objects in the domain of P - the two objects in the universe of type $[a : \alpha]$. Refer to Figure 11. Of the two objects only the object $[a := o_1]$ when bound to P makes the two expressions $P.a$ and o_1 semantically equivalent. We end up with the following substitution:

$$\{M = P, Q = [b := o_3], N = o_3, P = [a := o_1]\}$$

Since the body of Clause(2) is empty we are left with the empty resolvent. An empty resolvent indicates success. The substitution for M and N that makes $q_4(M, N)$ true is:

$$\{M = [a := o_1], N = o_3\}$$

The remaining solutions can be found by backtracking. We leave it to the reader to verify the other solutions.

4.6 Extending the Example to Illustrate Inheritance

We extend the example of the previous section to illustrate inheritance. Consider the universe shown in Figure 13 and the facts and rules shown in Figure 14.

Let

$$q_4(M, N)$$

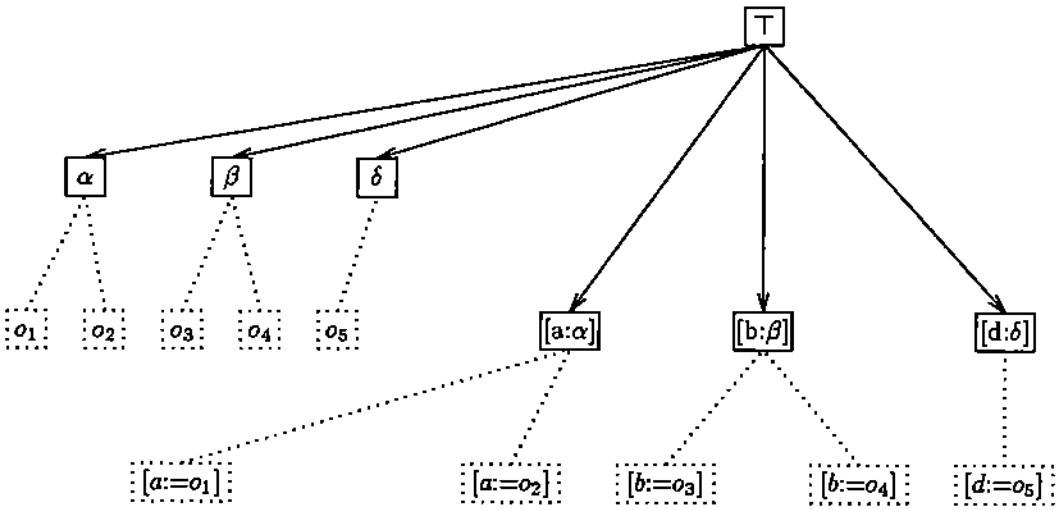

Figure 11: Universe of objects

be the query. Note that the logic variable M is not of type $[a : \alpha]$, but a subtype of it. The type of M is $[a : \alpha; e : \sigma]$. The entire search space of the query $q_4(M, N)$ is shown in Figure 14. The query yields only one substitution for M and N as solution:

$$\{M = [a := o_1; e := o_6], N = o_3\}$$

SEMLOG finds the above solution as follows. The goal $q_4(M, N)$ unifies with the head of Clause(4) $q_4(P, Q.b)$ yielding the following substitutions:

$$\{M_{[a:\alpha;e:\sigma]} = P_{[a:\alpha;e:\sigma]}, P_{[a:\alpha]} = P_{[a:\alpha;e:\sigma]}, Q = [b := o_3], N = o_3\}$$
$$\{M_{[a:\alpha;e:\sigma]} = P_{[a:\alpha;e:\sigma]}, P_{[a:\alpha]} = P_{[a:\alpha;e:\sigma]}, Q = [b := o_4], N = o_4\}$$

At this point it is important for the reader to compare the above substitutions with their counterparts in the previous example. Notice that besides type information there is also an additional binding in each of the above substitutions. The additional binding is $P_{[a:\alpha]} = P_{[a:\alpha;e:\sigma]}$. The reason for this additional binding is that since the logic variable M is of type $[a : \alpha; e : \sigma]$, unifying M and P has constrained P to a subtype of its original type. The type of P for all future references is $[a : \alpha; e : \sigma]$.

As before we can pursue either substitution for the next step. If we choose the second substitution, replace the goal $q_4(M, N)$ by the body of Clause(4), and make the appropriate substitutions we end up with the following resolvent:

$$q_1([b := o_4], [d := o_5]), q_2(P.a, [b := o_4])$$

Next, we try matching $q_1([b := o_4], [d := o_5])$ in the resolvent. The goal does not unify with the head of any of the clauses in our list of clauses. We have reached a dead end, so we backtrack and pursue the other substitution:

$$\{M_{[a:\alpha;e:\sigma]} = P_{[a:\alpha;e:\sigma]}, P_{[a:\alpha]} = P_{[a:\alpha;e:\sigma]}, Q = [b := o_3], N = o_3\}$$

and end up with the following resolvent:

$$q_1([b := o_3], [d := o_5]), q_2(P.a, [b := o_3])$$

This time the goal $q_1([b := o_3], [d := o_5])$ matches with the head of Clause(1). No new bindings are introduced as a result of the match, and since the body of Clause(1) is empty we have:

LOGIC VARIABLES			PREDICATE SIGNATURES
N, S	:	β	$q_1([b:\beta],[d:\delta])$
M, P, R	:	$[a:\alpha]$	$q_2(\alpha,[b:\beta])$
Q	:	$[b:\beta]$	$q_3([a:\alpha],\beta)$

FACTS & RULES

(1) $q_1([b{:=}o_3],[d{:=}o_5])$
(2) $q_2(o_1,[b{:=}o_3])$
(3) $q_2(o_2,[b{:=}o_3])$
(4) $q_3([a{:=}o_2],o_4)$
(5) $q_4(P,Q.b)$ <=> $q_1(Q,[d{:=}o_5]),q_2(P.a,Q)$
(6) $q_4(R,S)$ <=> $q_3(R,S)$

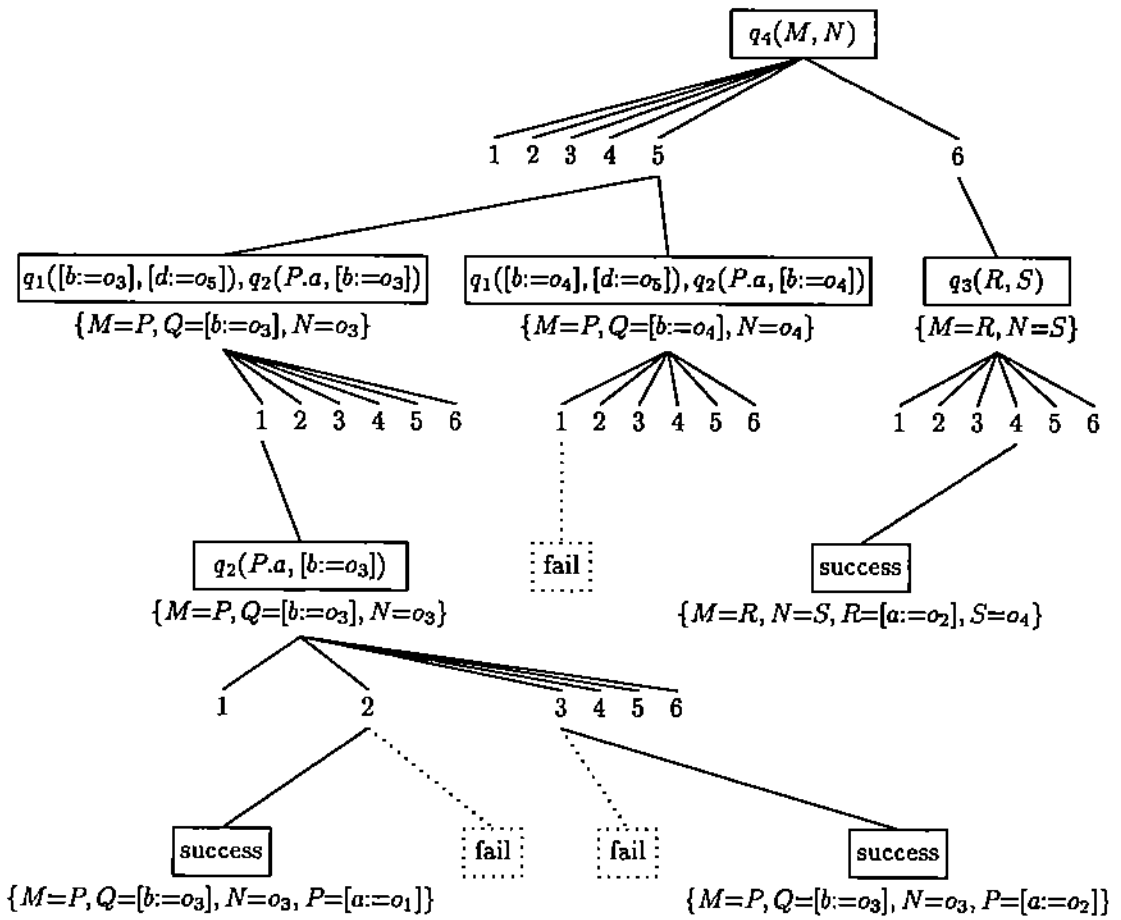

Figure 12: SEMLOG program and search space

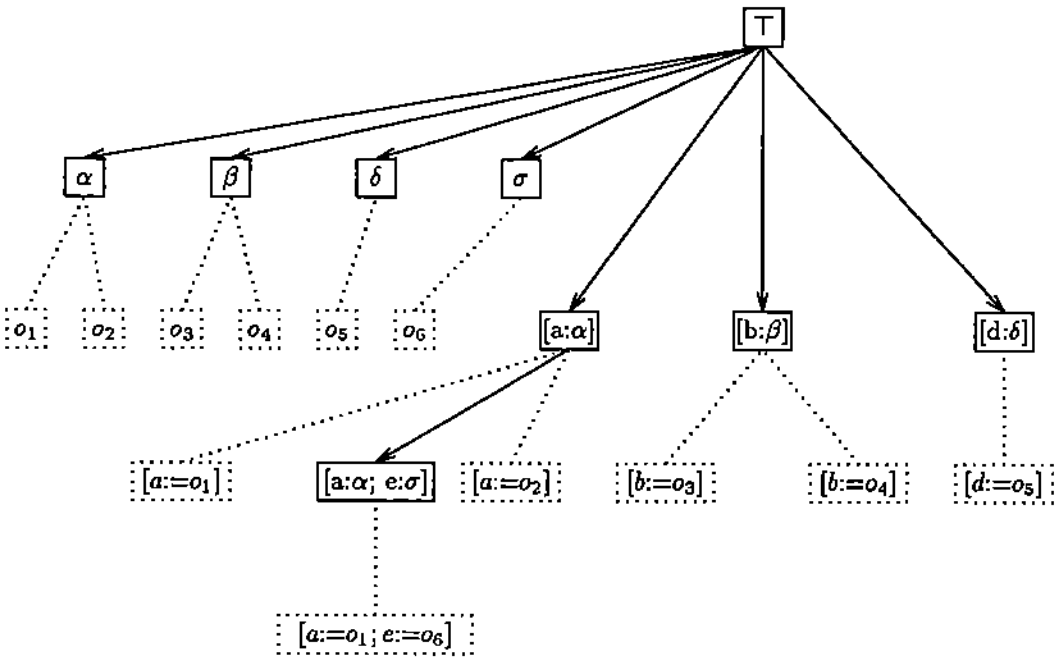

Figure 13: Universe of objects

$$q_2(P.a, [b := o_3])$$

as our new resolvent. We try to match $q_2(P.a, [b := o_3])$, with the head of Clause (2) $q_2(o_1, [b := o_3])$. Since P is not bound to anything in the current substitution, we need to find objects in the domain of P that when bound to P will render the two expressions $P.a$ and o_1 semantically equivalent. The type of P is $[a : \alpha; e : \sigma]$, not $[a : \alpha]$. There is only one object in the universe of type $[a : \alpha; e : \sigma]$. The object when bound to P makes the expressions $P.a$ and o_1 semantically equivalent. The new substitution is:

$$\{M_{[a:\alpha;e:\sigma]} = P_{[a:\alpha;e:\sigma]}, P_{[a:\alpha]} = P_{[a:\alpha;e:\sigma]}, Q = [b := o_3], N = o_3, P = [a := o_1; e := o_6]\}$$

We are then left with the empty resolvent. The substitution for M and N that makes $q_4(M, N)$ true is:

$$\{M = [a := o_1; e := o_6], N = o_3\}$$

5 Conclusions

This section includes a summary of the main contributions of this research. We discuss the approach that was adopted and the major accomplishments. Finally we outline the directions along which work can be pursued in the future to make SEMLOG a viable programming language for building knowledge-based systems.

5.1 Contributions

The primary objective of this research was to develop a programming language, one in which knowledge about a domain can be safely and concisely expressed. SEMLOG, provides the necessary primitives to accomplish the task of representing domain knowledge.

Domain knowledge has many forms, and in order to express these diverse forms of knowledge, we need a multiparadigm language, one that supports programming in many paradigms. If we were to use a language

LOGIC VARIABLES			PREDICATE SIGNATURES
N, S	:	β	$q_1([b:\beta], [d:\delta])$
P, R	:	$[a:\alpha]$	$q_2(\alpha, [b:\beta])$
Q	:	$[b:\beta]$	
M	:	$[a:\alpha; e:\sigma]$	

FACTS & RULES

(1) $q_1([b{:=}o_3], [d{:=}o_5])$
(2) $q_2([a{:=}o_1].a, [b{:=}o_3])$
(3) $q_2(o_2, [b{:=}o_3])$
(4) $q_4(P, Q.b) <=> q_1(Q, [d{:=}o_6]), q_2(P.a, Q)$

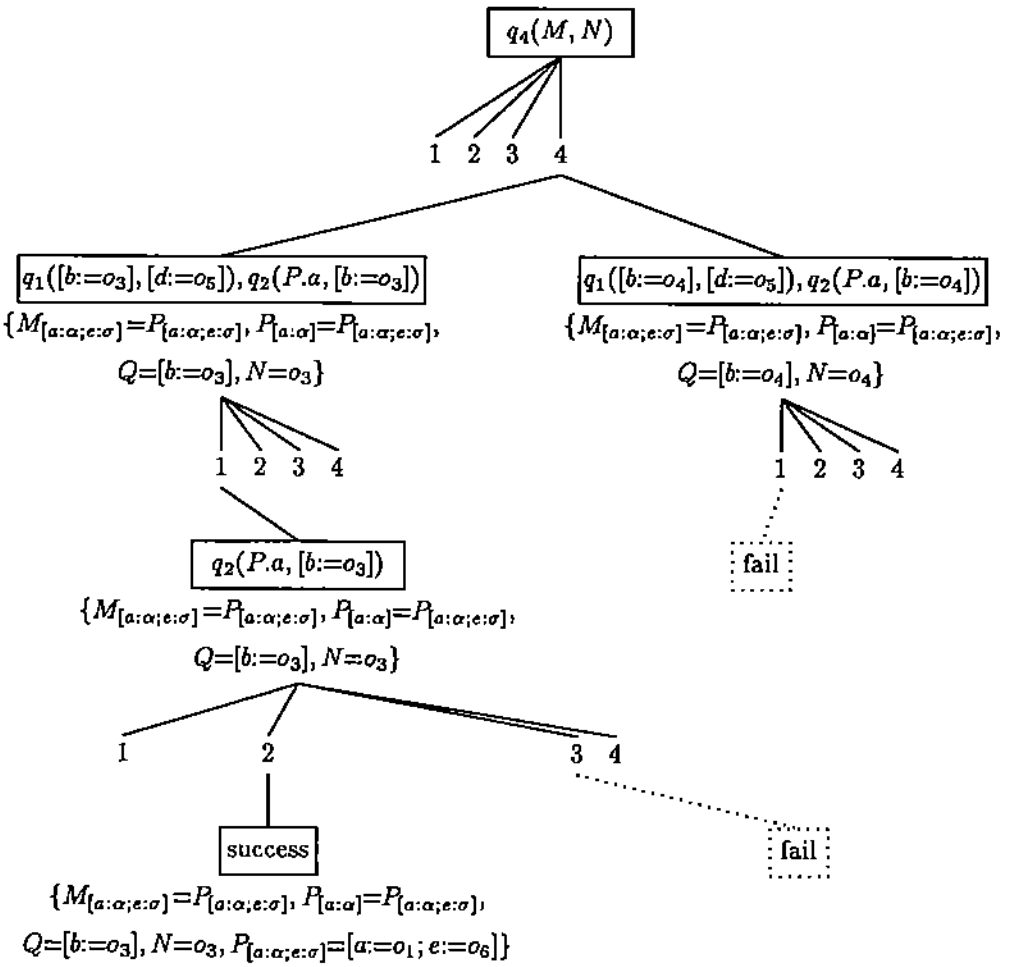

Figure 14: SEMLOG program and search space

that supports only one paradigm, we would have to resort to elaborate and obscure techniques to capture knowledge.

In the Turing machine sense, all common programming languages are universal. However, different paradigms allow for different things to be stated concisely. Some forms of knowledge can be stated more "naturally" and concisely in one paradigm than in another.

Several other researchers have also argued for the need to integrate the paradigms. Numerous multi-paradigm environments (not languages) were built; the most notable ones being LOOPS [Bobrow 1983] and KEE [Fikes 1985].

There are two important differences between our approach and that of the others. We do not embed one or more paradigms on top of another. Instead we identify the primitive language features that are necessary for representing knowledge about a domain. We then introduce these features in their minimal and essential form in a new language. The reason for defining a new language was because we were interested in studying the semantics of combining the paradigms.

Another important difference is that, unlike the others, we were interested in strong typing. We feel that support for a paradigm in a language comes not only in the obvious form of language primitives that allow programming in the paradigm, but also in the more subtle form of type checking to ensure that unintentional deviations from the paradigm are detected. The earlier the detection of these deviations the better. Early detection, type checking at compile time, guarantees that certain kinds of errors will not occur during program execution. It also enforces a programming discipline on the programmer.

SEMLOG provides powerful primitives for representing domain knowledge. It provides primitives for describing: domain objects and their attributes, the taxonomic arrangement of domain objects, the association of one or more objects in a relationship, and rules for making decisions. But, more importantly it provides the semantics of combining these primitives.

5.2 Future Work

Although the language provides many powerful primitives for capturing knowledge, in its present form it is still bare. Many constructs are needed in the language to make the programming task easier. Most of these constructs can be added to the language without significantly changing the semantics of the language. Others, however, may have an impact on the semantics and this remains to be explored.

One issue that is bound to have a significant impact on the semantics of the language is the inclusion of imperative features. Imperative features are needed in the language to model state transformations in the domain. State transformations occur in the domain when: new entities are introduced, existing entities disappear, the attributes of the entities change, and when new relations among objects are formed. To model these state transformations we need in the language operators for changing the set of clauses in the knowledge base (the equivalent of PROLOG's assert operator), and a type-object lattice that could possibly change during querying.

The language in its present form does not have any imperative features. The state of the domain is specified prior to querying. It remains unchanged during the actual querying process.

Some of the directions in which the language can be extended is outlined below; the list is by no means complete.

1. The incorporation of type constructors such as cartesian products, lists, arrays, etc., would certainly make the language more convenient to program in.

2. Presently the only forms of queries in the language are queries for which the system responds with all the solutions. Variations of this can be incorporated — queries for which the system responds with only one solution, queries for which the system generates one solution at a time on demand.

3. Another important extension would be a default mechanism capability. Presently there are no mechanisms for objects to inherit default values when they are created; all the object attributes must be specified.

4. Imperative features are needed in the language to model state transformations in the domain.

References

[Aït-Kaci 1986a] H. Aït-Kaci, R. Nasr, *"Logic and Inheritance,"* in Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, pp. 219–228, 1986

[Aït-Kaci 1986b] H. Aït-Kaci, R. Nasr, *"LOGIN: A Logic Programming Language with built-in Inheritance,"* Journal of Logic Programming, No. 3, pp. 185–215, 1986

[Albano 1985] A. Albano, L. Cardelli, R. Orsini, *"Galileo: A Strongly-Typed, Interactive Programming Language,"* ACM Trans. on Database Syst., Vol. 10, No. 2, pp. 230–260, June 1985

[Bobrow 1977] D. G. Bobrow, T. Winograd, *"An Overview of KRL: A Knowledge Representation Language,"* Cognitive Science, Vol. 1, No. 1, pp. 3–46, 1977

[Bobrow 1983] D. G. Bobrow, M. Stefik, *"The LOOPS Manual,"* Technical Report, Xerox PARC, 1983

[Bobrow 1985] D. G. Bobrow, *"If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms,"* IEEE Trans. on Software Eng., Vol. SE-11, pp. 1401–1408, November 1985

[Bonczek 1981] R. H. Bonczek, C. W. Holsapple, A. B. Whinston, *"Foundations of Decision Support Systems,"* Academic Press, 1981

[Borgida 1985] A. Borgida, *"Features of Languages for the Development of Information Systems at the Conceptual Level,"* IEEE Software, pp. 63–72, January 1985

[Brachman 1983] R. J. Brachman, *"What IS-A is and isn't: An Analysis of Taxonomic Links in Semantic Networks,"* Computer, Vol. 16, No. 10, pp. 30–36, October 1983

[Buchanan 1978] B. G. Buchanan, E. A. Feigenbaum" *"DENDRAL and Meta-DENDRAL: Their Application Dimensions,"* Artificial Intelligence, Vol. 11, pp. 5–24, 1978

[Cardelli 1984] L. Cardelli, *"A Semantics of Multiple Inheritance,"* in Semantics of Data Types, Lecture Notes in Computer Science, No. 173, pp. 51–67, Springer-Verlag 1984

[Clancey 1979a] W. J. Clancey, *"Tutoring Rules for Guiding a Case Method Dialogue,"* International Journal of Man-Machine Studies, Vol. 11, pp. 25–49, 1979

[Clancey 1979b] W. J. Clancey, E. H. Shortliffe, B. G. Buchanan, *"Intelligent Computer-Aided Instruction for Medical Diagnosis,"* in Proceedings of the Third Annual Symposium on Computer Applications in Medical Care, pp. 175–183, 1979

[Clocksin 1984] W. F. Clocksin, C. S. Mellish, *"Programming in Prolog,"* Springer-Verlag, 1984

[Dahl 1966] O. J. Dahl, K. Nygaard, *"SIMULA—an Algol-Based Simulation Language,"* CACM, Vol. 9, pp. 671–678, 1966

[Duda 1979] R. O. Duda, J. G. Gaschnig, P. E. Hart, *"Model Design in the PROSPECTOR Consultant System for Mineral Exploration,"* in Expert Systems in the Micro-Electronic Age, ed., D. Michie, Edinburgh University Press, pp. 153–167

[Fagan 1979] L. M. Fagan, J. C. Kunz, E. A. Feigenbaum" J. Osborn, *"Representation of Dynamic Clinical Knowledge: Measurement Interpretation in the Intensive Care Unit,"* in IJCAI, Vol. 6, pp. 260–262, 1979

[Feigenbaum 1971] E. A. Feigenbaum" B. G. Buchanan, J. Lederberg, *"On Generality and Problem Solving: A Case Study using the DENDRAL Program""* in Machine Intelligence, Vol. 6, eds., B. Meltzer and D. Michie, Edinburgh University Press, pp. 165–190

[Feigenbaum 1977] E. A. Feigenbaum" *"The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering,"* in IJCAI, Vol. 5, pp. 1014–1029, 1977

[Fikes 1985] R. Fikes, T. Kehler, *"The Role of Frame-Based Representation in Reasoning,"* CACM, Vol. 28, No. 9, pp. 904–920, September 1985

[Forgy 1981] C. L. Forgy, *"OPS5 User's Manual,"* Technical Report, CMU-CS-81-135, Dept. of Computer Science, Carnegie-Mellon University, July 1981

[Goguen 1984] J. A. Goguen, J. Meseguer, *"Equality, Types, Modules, and Generics for Logic Programming,"* Second International Logic Programming Conference, Uppsala University, Sweden, July 1984

[Goldberg 1983] A. Goldberg, D. Robson, *"Smalltalk-80: The Language and its Implementation,"* Addison-Wesley, 1983

[Gullichsen 1985] E. Gullichsen, *"BiggerTalk: Object-Oriented PROLOG,"* Technical Report STP-125-85, MCC Software Technology Program" 9430 Research Blvd., Austin, Texas 78759, 1985

[Hammer 1981] M. Hammer, D. McLeod, *"Database Description with SDM: A Semantic Data Model,"* ACM Trans. on Database Syst., Vol. 6, No. 3, September 1981

[Kent 1979] W. Kent, *"Limitations of Record-Based Information Models,"* ACM Trans. on Database Syst., Vol. 4, No. 1, pp. 107–131, March 1979

[Kowalski 1974] R. Kowalski, *"Predicate Logic as a Programming Language,"* Proceedings of the IFIP 74, pp. 569–574, 1974

[Kowalski 1979] R. Kowalski, *"Algorithm = Logic + Control,"* CACM, Vol. 22, No. 7, pp. 424–431, July 1979

[Lindsay 1980] R. K. Lindsay, B. G. Buchanan, E. A. Feigenbaum" J. Lederberg, *"Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project,"* McGraw-Hill, 1980

[Martin 1971] W. A. Martin, R. J. Fateman, *"The MACSYMA System""* in Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, pp. 59–75, 1971

[McDermott 1980] J. McDermott, *"R1: An Expert System in the Computer System Domain,"* in AAAI, Vol. 1, 1980

[Milner 1984] R. Milner, *"A Proposal for Standard ML,"* Proceedings of the ACM Symposium on Lisp and Functional Programming, 1984

[Minsky 1975] M. A. Minsky, *"A Framework for Representing Knowledge,"* in The Psychology of Computer Vision, ed., P. Winston, New York: McGraw-Hill, 1975

[Mylopoulos 1980] J. Mylopoulos, P. A. Bernstein, H. K. T. Wong, *"A Language Facility for Designing Database-Intensive Applications,"* ACM Trans. on Database Syst., Vol. 5, No. 2, pp. 185–207, June 1980

[Newell 1981] A. Newell, *"The Knowledge Level,"* Artificial Intelligence Magazine, Vol. 2, No. 2, pp. 1–20, Summer 1981

[Robinson 1965] J. A. Robinson, *"A Machine-Oriented Logic Based on the Resolution Principle,"* J. ACM, Vol. 12, No. 1, pp. 23–41, January 1965

[Schmidt 1980] J. W. Schmidt, M. Mall, *"PASCAL/R Report,"* Report No. 66, Fachbereich Informatik, University of Hamburg, January 1980

[Shortliffe 1976] E. H. Shortliffe, *"Computer-Based Medical Consultation: MYCIN,"* American Elsevier, 1976

[Stabler 1986] E. P. Stabler Jr., *"Object-Oriented Programming in PROLOG,"* AI Expert, October 1986

[Stefik 1979] M. Stefik, *"An Examination of a Frame-Structured Representation System""* in IJCAI, Vol. 6, pp. 845–852, 1979

[van Melle 1979] W. van Melle, *"A Domain-Independent Production-Rule System for Consultation Programs,"* in IJCAI, Vol. 6, pp. 923–925, 1979

[Weinreb 1981] D. Weinreb, D. Moon, *"Lisp Machine Manual,"* Symbolics Inc., 1981

[Zaniolo 1984] C. Zaniolo, *"Object-Oriented Programming in PROLOG,"* Proceedings of the 1984 IEEE Symposium on Logic Programming, pp. 265–270 1984